# Error Localization for Sequential Effect Systems

Colin S. Gordon[0000−0002−9012−4490] and Chaewon Yun

{csgordon,cy422}@drexel.edu
Drexel University, Philadelphia PA 19104, USA

**Abstract.** We describe a new concrete approach to giving predictable error locations for sequential (flow-sensitive) effect systems. Prior implementations of sequential effect systems rely on either computing a bottom-up effect and comparing it to a declaration (e.g., method annotation) or leaning on constraint-based type inference. These approaches do not necessarily report program locations that precisely indicate where a program may "go wrong" at runtime.

Instead of relying on constraint solving, we draw on the notion of a residual from literature on ordered algebraic structures. Applying these to effect quantales (a large class of sequential effect systems) yields an implementation approach which accepts exactly the same program as an original effect quantale, but for effect-incorrect programs is guaranteed to fail type-checking with predictable error locations tied to evaluation order. We have implemented this idea in a generic effect system implementation framework for Java, and report on experiences applying effect systems from the literature and novel effect systems to Java programs. We find that the reported error locations with our technique are significantly closer to the program points that lead to failed effect checks.

## 1   Introduction

Effect systems are a well-established technique for extending a base type system that reasons about input and output shapes and available operations, to also statically reason about behaviors of code. However, error reporting for effect systems has not been systematically studied. Existing implementations of effect systems report errors in one of two ways.

The classic approach is checking that each individual operation's effect is less than some bound [6,68,58,32,28] and reporting errors for any individual operation whose check fails. For example, this is how Java's checked exceptions are handled: every possibly-throwing expression's `throws` clause is checked against that of the enclosing method. This yields highly precise error locations (e.g., reporting specific problematic method invocations or `throw` statements), but applies only for the (common) case of flow-insensitive effect systems.

The more general approach is to raise an error for whatever program point gave rise to a failing constraint during type inference [2,49,8,34,52,35,36,65], which works for a wide variety of effect systems. However this leads to the well-known difficulty with localizing mistakes from type inference errors: the constraint which failed may be far away from an actual programmer mistake.

Technically a third possibility is to compare the computed effect of a method body against an annotated or assumed bound. This can work for any effect system, but we know of no implementations taking this approach, which would yield highly imprecise error messages (basically, "this method has an effect error").[1]

This is an unforunate state of affairs: less powerful effect systems have localized error reporting (the first approach), while more powerful flow-sensitive effect systems — arguably in more dire need of precise error reporting — are stuck with error reports that are unpredictable (approach 2) or maximally imprecise (approach 3). This paper shows how to derive precise, predictable error reporting for flow-sensitive effect systems as well, by noticing that the first approach is in fact an error reporting optimization of the third: applied to the same effect system, they accept exactly the same programs, but while the third directly implements typical formalizations, the first in fact cleverly exploits algebraic properties of the third for more precise error reporting. By articulating and generalizing these properties, we obtain a new more precise error reporting mechanism for sequential [66] effect systems.

In the common case of an effect system where effects are partially ordered (or pre-ordered), while type-and-effect checking code with a known upper bound (such as a Java method with a throws clause), it is sufficient to check for each operation (e.g., method invocation or throw statement) whether the effect (e.g., the possibly-thrown checked exceptions) is less than the upper bound (e.g., `throws` clause). If not, an error is reported for that operation. This is a deviation from how such effect systems are typically formalized, which is as a join semilattice, where formally the error would not occur until the least upper bound over *all* subexpressions' effects was compared against the declared bound for the code. Directly implementing this typical formalization is sound, but for a large method provides no direct clue as to where the problematic code may be in the method.

This paper explores in detail why this optimization is valid, and uses that insight to generalize this precise error reporting to *sequential* effect systems. While the validity of this switch from joins in metatheory to local ordering checks in implementations is intuitively clear given basic properties of joins (namely, $\forall X, b. ((\bigsqcup X) \sqsubseteq b) \Leftrightarrow (\forall a \in X. a \sqsubseteq b))$, it is not obvious that a corresponding transformation exists that can be applied to *sequential* effect systems to move from global error checking to incremental error checking. Our contributions include:

- We explicitly identify and explain the common pattern of formalizing an effect system using operators that compute effects, while implementing the systems differently. We explain why this is valid.
- We generalize this to arbitrary sequential effect systems characterized as effect quantales.
- We describe an implementation of this approach for Java programs in the first implementation framework for sequential effect systems for Java.

---

[1] Some implementations are designed for effect inference to always succeed, with a secondary analysis rejecting some effects outside the effect system [63,62], and others with unavailable source do not provide enough detail to ascertain their effect checking algorithm [20,7,18,19,59].

– We describe experiments implementing sequential effect systems in this framework and applying them to real Java programs, arguing that this theoretically grounded approach yields precise errors.

## 2   Background

Effect systems extend traditional type systems with information about side effects of program evaluation, which can be tailored to program behaviors of interest. Applications have included analyzing what regions of memory are accessed [46,24,67]; ensuring data race freedom [17,6,15,1], deadlock freedom [64,16,29,35], or other more targeted concurrency safety properties like safe use of GUI primitives [28,68,45]; checking atomicity in concurrent programs [21,20]; checking safety of dynamic software updates [48]; checking communication properties [2,49]; dataflow properties [36,4] or general safety properties of execution traces [63,61,38].

Effect systems extend the typing judgement to include an additional component, the *effect*, which is a syntactic description of an upper bound on an expression's behavior. The typical judgment form $\Gamma \vdash e : \tau \mid \chi$ is interpreted as meaning the under variable typing assumptions $\Gamma$, evaluating expression $e$ will produce a result of type $\tau$ (if execution terminates), exhibiting at most behaviors described by $\chi$. Function types are also extended in effect systems to carry a *latent* effect $\chi$, typically written superscript above a function arrow, as in $\tau \xrightarrow{\chi} \tau'$, indicating that $\chi$ is a bound on the function body's behavior, which the type rule for function application incorporates into the effect of function invocation.

Most often the (representations of) behaviors an effect system reasons about are assumed to form a join semilattice, and intuitively the effect of an expression is then the least upper bound (join) of the effects of all (executed) subexpressions. But this model of effect systems, while broad and including many useful and powerful effect systems, is incomplete. Many of the effect systems of interest [64,29,49,2,63,63,38,21,20,35,48,36,4] have additional structure because they track behaviors sensitive to evaluation order, and therefore have not only a partial ordering on effects to model a notion which behaviors subsume others, but also a notion of sequencing effects to track ordering of behaviour.

Consider using effects to determine where other threads might observe intermediate results of a method in a concurrent program [20,21,72]. There `lock(x);...unlock(x);` is clearly different from `unlock(x);...lock(x);` — in the former case, no other thread in a data-race-free program can observe intermediate states of the ellided section, while in the latter case, another thread could acquire lock `x` and observe changes to the state it protects. *Sequential* effect systems like Flanagan and Qadeer's [20,21] add an additional operator to effect systems, $\rhd$ in this paper, to model effects occurring in a specific order. $\sqcup$ is retained as the least-upper-bound of effects (e.g., for conditionals). So if `lock(x);` has effect $R$ and `unlock(x)` has effect $L$ (for reasons elaborated in Section 3.3), this permits the two code fragments to have different effects due to their different orders: $R \rhd L = A$ (atomic), while $L \rhd R = \top$ (non-atomic). This follows the standard type rules for sequential effect systems (the black portions

of Figure 3), where some uses of $\sqcup$ in classic effect systems are replaced by $\triangleright$ where program evaluation order is known.

There is still some active debate as to what the appropriate common model of these *sequential* effect systems should be. They are captured most generally by Tate's effectoids [66] (or equivalently, by polymonads [34]), but these are typically acknowledged to be more general than most systems require. More pragmatic proposals include graded monads [37] and effect quantales [25,27], which differ primarily in what kinds of distributive laws are assumed (or not assumed) for how least-upper-bound and sequencing interact. Gordon [27] gives a detailed survey of general models of sequential effect systems, and their relationships.

We work with sequential effect systems characterized by effect quantales, because their application to mainstream programming languages seems furthest-developed, including general approaches to deriving from a basic effect quantale treatments of loops [27] and constructs derivable from tagged delimited continuations [26] (e.g., exceptions, generators); for related frameworks only single examples exist. Gordon [27] also gives a survey of how a wide range of specific sequential effect systems from the literature are modeled by effect quantales.

**Definition 1 (Effect Quantale [27]).** *An* effect quantale *is a structure* $Q = \langle E, \sqcup, \triangleright, I \rangle$ *composed of:*

- *A set of effects (behaviors) $E$*
- *A* partial *join (least-upper-bound) $\sqcup : E \times E \rightharpoonup E$*
- *A* partial *sequencing operator $\triangleright : E \times E \rightharpoonup E$*
- *A unit element $I$*

*such that*

- *$\langle E, \triangleright, I \rangle$ is a partial monoid with unit $I$ (i.e., $\triangleright$ is an associative operator with left and right unit $I$)*
- *$\langle E, \sqcup \rangle$ is a partial join semilattice (i.e., $\sqcup$ is commutative, associative, and idempotent)*
- *$\triangleright$ distributes over $\sqcup$ on both sides*
    - *$x \triangleright (y \sqcup z) = (x \triangleright y) \sqcup (x \triangleright z)$*
    - *$(x \sqcup y) \triangleright z = (x \triangleright z) \sqcup (y \triangleright z)$*

Note that when writing relations involving possibly-undefined expressions (e.g., since $x \sqcup y$ may be undefined), we consider two expressions equal if they both evaluate to the same element of $E$, or are both undefined.[2]

From the partial join we can derive a partial order on effects: $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$. Again we must specify the meaning of $\sqsubseteq$ on possibly-undefined expressions: $x \sqsubseteq y$ is defined only if the join of $x$ and $y$ is defined.

---

[2] Readers who tire of thinking about partiality can approximate this by imagining there is an additional distinguished error element, greater than all others, and preserved by all operators, representing undefined results, and which is invalid in later type rules (as in the original axiomatization [25]).

Both $\triangleright$ and $\sqcup$ are monotone in both arguments, in the sense that if $a \sqsubseteq b$ and $x \sqsubseteq y$, then $a \triangleright x \sqsubseteq b \triangleright y$ (when the right side is defined) and similarly for $\sqcup$.

Note that any join-semilattice-based effect system can be modeled in this system, using $\perp$ for identity, and reusing join for sequencing as well.

Gordon [27] also describes properties of a partial unary iteration operation $-^*$ used to characterize loop effects, guaranteed to be:

$$
\begin{aligned}
\text{Extensive } &\forall x.\, x \sqsubseteq x^* \\
\text{Idempotent } &\forall x.\, (x^*)^* = x^* \\
\text{Monotone } &\forall x, y.\, x \sqsubseteq y \Rightarrow x^* \sqsubseteq y^* \\
\text{Foldable } &\forall x.\, (x^*) \triangleright (x^*) \sqsubseteq x^* \\
\text{Possibly-Empty } &\forall x.\, I \sqsubseteq x^*
\end{aligned}
$$

An optimally precise iteration operator can be derived for most effect quantales of interest [27]: all finite effect quantales, and all effect quantales which have finite meets of elements greater than the unit element. This includes all prior specific effect systems considered in the survey section of that paper.

## 2.1   Implementing Effect Systems

We concern ourselves with implementing effect system checking in a setting where the expected effect of every method in the program is given, rather than inferred. This models a reasonable integration of effects into languages that require explicit method signatures, consistent with a number of prior implementations of effect systems [68,28,32,20]. Experience has shown that while full inference has value, for many effect systems, a reasonable default (or local customization of defaults) is often sufficient to achieve modest annotation overhead. This also models scenarios where full inference has been employed, but manual annotations are required to refine undesirable inferences and force type checking to produce errors in a method that is intended to have a certain effect, but was inferred to have an incompatible effect. We speak of methods because our prototype (Section 5) targets Java, but the same principles would apply to procedural or functional languages.

*Global Reporting* Many formalizations of effect systems use a join-semilattice or effect-quantale-like formulation of the system, so an implementation can generally compute the effect of an expression bottom-up. The result is that for code with a fixed bound $\chi$ (say, the declared effect of a method), the implementation uses the $\sqcup$ and/or $\triangleright$ operators to compute the body effect $\chi'$, and then once for each method checks that $\chi' \sqsubseteq \chi$. This is a direct implementation of common metatheory for effect systems, but as the only effect *check* occurs at the granularity of entire method or function bodies, there is only one possible error location for such a technique to report: the entire method or function body. For example, consider locating the error in Figure 1, as studied in prior work [20]. This approach would use type rules like those of Figure 3 to compute the method body's effect, which would be $\top$ (non-atomic). Only then would this be checked against the method annotation that this method should be atomic, and type-checking would fail. At

```
1   @Atomic
2   @ThrownEffect(exception = StringIndexOutOfBoundsException.class,
3                 behavior = Atomic.class)
4   public synchronized StringBuffer append(StringBuffer sb) {
5     if (sb == null) {
6         sb = NULL;
7     }
8     int len = sb.length(); // Already atomic
9     int newcount = count + len;
10    if (newcount > value.length)
11      expandCapacity(newcount);
12    sb.getChars(0, len, value, count); // <-- Error: also atomic
13    count = newcount;
14    return this;
15  }
```

Fig. 1: Excerpt from JDK1.4 `StringBuffer` implementation.

that point, all information that line 12 has critically contributed to the error (because it attempts a second atomic action without synchronization) has been lost, leading to lack of precision in error reporting. We are unaware of any concrete systems that explicitly acknowledge implementing this approach, though language in some papers is suggestive of such an approach (e.g., mentioning that "inspection of a method" with an error revealed a problem, as opposed to indicating an error was flagged on a specific line of code).

*Precise Reporting for Commutative Effect Systems* Most effect system implementations are limited to join-semilattice or partial order structures, which ignore program order. Of those with available implementations, all we know of give precise error locations for code expressions which would lead to failing effect system checks by exploiting the trick mentioned in the introduction, that this class of system permits checking effects incrementally by checking if each subexpression's effect is less than the bound, since this is equivalent to computing the join of subexpression effects and comparing to the bound. This is true of all available implementations we are aware of, including Java's checked exceptions [32], the modern implementation of Gordon et al.'s UI effects [28] in the Checker Framework [53,13], Toro and Tanter's framework [68] for gradual effects [60], Rytz et al.'s work for Scala [58], Deterministic Parallel Java [6], and others. As mentioned earlier, this trick does not work for general sequential effect systems.

*Constraint-Based Reporting* Most prior implementations of sequential effect systems —— and *all* prior implementations *frameworks* for sequential effect systems [34,8,9,10,52] — use type inference to infer effects. This results in the standard trade-offs for global constraint-based type inference: types (and effects) are inferred with low developer effort when possible, but errors can be cryptic, and implicate program locations unrelated to the error. In particular, these implementations tend to generate subtyping (and subeffecting) constraints from

the program structure, which are then solved incrementally by a fixpoint solver. Errors are reported at the location corresponding to the first constraint which is found to be inconsistent. However, that constraint may be totally unrelated to any problematic statement. Consider the brief JavaScript program

```
var x = 3; var y = x; requiresString(y)
```

It is possible for a constraint solver to flag any of the three statements as a type error, assuming the invoked method is typed as requiring a string input. Flagging the first or third lines is reasonable, as they are the sources of the contradiction. However, solvers are permitted to report the middle line as erroneous as well (for storing a number into a string-containing variable), which is not terrible in this case, but becomes problematic with larger blocks of code. This has inspired a wealth of work on various techniques to reduce or partially compensate for (but not eliminate) this unpredictability [54,55,44,33,42,11]. In principle such work is applicable to existing approaches to inferring effects in sequential effect systems [19,52,50], but unpredictability would remain. While details vary between systems, in general this approach could flag any line of code in Figure 1 as erronneous.

## 3   Local Errors for Sequential Effect Systems

We would like to obtain precise error reporting for sequential effect systems in general. Because effect quantales subsume the join semilattice model of effect systems [27], we can hope to draw some inspiration from the corresponding optimization on traditional commutative effect systems: that optimization should be a special case of a general solution.

Let us fix an expression $e$ whose effect we would like to ensure is less than $\chi$. Let us assume a set $\{\chi_i \mid i \in \mathsf{Subterms}(e)\}$ where $\chi_i$ is the static effect of subterm $i$ from a bottom-up effect synthesis. For now, we will assume all such effects are defined (i.e., that the bottom-up synthesis of effects never results in undefined effects). For the case where effects form a join-semilattice, and all bottom-up computation is joins (no other operators play a role), we can formally relate the global and precise implementations of join-semilattice effect systems by observing

$$\bigsqcup\{\chi_i \mid i \in \mathsf{Subterms}(e)\} \sqsubseteq \chi \Leftrightarrow \forall i \in \mathsf{Subterms}(e).\, \chi_i \sqsubseteq \chi$$

as suggested in the introduction. The left side of the iff expresses the global view that the join of all subexpression effects must be bounded by $\chi$. The right side expresses the local view that each individual subexpression's effect must be less than the bound $\chi$. (This formulation suggests some redundant checks; we return to this later.) One way to express the intuition behind this formula is that performing all of the local checks corresponds to ensuring that each individual $\chi_i$ can be further combined with some other effects (here, by join), and the result will still be bounded by $\chi$. Conversely, if there exists some $j \in \mathsf{Subterms}(e)$ such that $\chi_j \not\sqsubseteq \chi$, then no combination with other effects can yield something satisfying the bound $\chi$.

We dub this informal characterization as the notion of *completing* an effect. An effect $\chi_i$ *can be completed to* $\chi$ if there exists some effect $\chi'$ such that the combination of $\chi_i$ and $\chi'$ is $\sqsubseteq \chi$. The general intuition is that if $\chi_i$ can be completed to $\chi$, it is possible to "add more behaviors" to $\chi_i$ and obtain an effect less than $\chi$, in the sense that it is possible to extend a program with effect $\chi_i$ with additional behaviors such that the overall effect is less than $\chi$. We formalize this later in this section, such that the result will allow precise reporting of error locations, such as line 12 in Figure 1. But first we must recall and customize a bit of relevant math.

### 3.1   Residuals

The literature on ordered semigroups [5] and non-commutative substructural logics [41,23] contains many applications of the notion of a *residual* [71,14]:

**Definition 2 ((Right) Residual).** *A (right)* residual *operation on an ordered monoid M is a binary operation* $- \backslash - : M \times M \to M$ *such that for any x, y, and z,* $x \leq y \backslash z \Leftrightarrow y \cdot x \leq z$

That is, the right residual $y \backslash z$ of $z$ by $y$ (also read as $y$ under $z$) is an element of $M$ such that, when sequenced *to the right* of $y$, yields an element no greater than $z$ (but definitely ordered $\leq z$).

We can adapt this for effect quantales as well:

**Definition 3 ((Right) Quantale Residual).** *A (right) residual operation on an effect quantale Q is a partial binary operation* $- \backslash - : Q \times Q \rightharpoonup Q$ *such that for any x, y, and z,* $x \sqsubseteq y \backslash z \Leftrightarrow y \triangleright x \sqsubseteq z$. *A* (right) residuated effect quantale *is an effect quantale with a specified choice of (right) residual operation.*

Consider the case of type-checking $e_1; e_2$, where $\Gamma \vdash e_1 : \chi_1$ and $\Gamma \vdash e_2 : \chi_2$, and ensuring that the effect of the sequential composition of these expressions — $\chi_1 \triangleright \chi_2$ — is bounded by $\chi$. A residual on effects can tell us if this is *possible* based on analyzing *only* $e_1$, in some cases rejecting programs before even analyzing $e_2$.

If $\chi_1 \backslash \chi$ is undefined, then there *does not exist* $\chi'$ such that $\chi_1 \triangleright \chi' \sqsubseteq \chi$. If there were some such $\chi'$, then by the definition of the residual operation, $\chi' \sqsubseteq \chi_1 \backslash \chi$, which would imply the residual was defined. Thus an implementation could eagerly return an error after synthesizing the effect $\chi_1$ for $e_1$, and determining the residual was undefined. This is a subtle point about the definition above: it states not only properties of the residual when it is defined, but also requires it to be defined in certain cases.

Many effect quantales have a right residual in the sense of Definition 2, but not all do, and since effect systems are primarily concerned with sound bounds on behavior rather than exact characterizations of behavior, we actually require only a slightly weaker variant of residual:

**Definition 4 (Weak (Right) Quantale Residual).** *A weak (right) residual operation on an effect quantale Q is a partial binary operation* $- \backslash - : Q \times Q \rightharpoonup Q$ *such that for any x, y, and z:*

- *Residual bounding: $x \sqsubseteq y \setminus z \Rightarrow y \rhd x \sqsubseteq z$*
- *Residual existence: $y \rhd x \sqsubseteq z \Rightarrow \exists r. r = y \setminus z$*
- *Self-residuation: $\exists r. z \setminus z = r$*
- *Unit residuation: $I \setminus z = z$*

*A* (right) residuated effect quantale *is an effect quantale with a specified choice of weak (right) residual operation.*

This is the notion of residual we work with, and this weakening is necessary to capture aspects of non-local control flow [26]. Every residual operation in the remainder of this paper is a weak residual, though for brevity we simply refer to them as residuals. The *total* residual (Definition 3) implies the axioms of the weak residual, so in some cases we present a total residual as a weak residual.

We take this weak right residual to be our formal notion of completion: an effect $\chi$ can be completed to $\chi'$ if the weak residual $\chi \setminus \chi'$ is defined.

It is worth noting that the literature also contains a definition of left residual, which we could use to similarly issue an eager warning given only $\chi$ and $\chi_2$. However, notice that the right residual corresponds to type-checking traversals proceeding in the standard left-to-right evaluation order standard in (most) languages using call-by-value reduction. Because most other analysis tools, and in practice most developer investigation of program-order dependent behaviors proceeds in tandem with evaluation order, we focus solely on the right residual. However, all results in the rest of the paper can be dualized to the left residual. Because we focus exclusively on the right residual, for the rest of the paper we will drop the qualifier "right" and simply refer to unqualified residuals.

This seems a promising approach, but detailing it fully requires also connecting our notion of completion to the way effects are actually combined during type-checking (e.g., most programs are not basic blocks of primitive actions). Before doing so, we build further intuition by describing the residual operations for a few existing effect quantales in the literature, based on Gordon's formalization [27].

### 3.2   Residual Examples

This section gives several examples of residuated effect quantales, to show that many existing effect systems already naturally satisfy the requirements of our weak residual, so while it is not mathematically the case that all effect quantales have a weak residual, it appears known effect systems typically do. Our extended techincal report [31] contains additional examples.

**Traditional Commutative Effects**  In the case where the effect quantale is simply a (partial) join semilattice (so $\rhd = \sqcup$), the residual is simply:

$$X \setminus Y = Y \quad \text{when} \quad X \sqsubseteq Y$$

Thus the residual is exactly the local subeffect comparison performed by local implementations of effect checking.

**Formal Languages and Quotients Thereof** As formal languages can model sets of acceptable behaviors (and are often used for this purpose, most frequently in automata-theoretic model checking), it is worth considering formal languages as effects. Indeed, languages of finite words over a finite alphabet form an effect quantale: Here we recall a distillation of a number of general behavioral trace effect systems [63,61,38] given by Gordon [27]:

**Definition 5 (Finite Trace Effects).** *Effects tracking sets of finite event traces, for events in an alphabet $\Sigma$, form an effect quantale:*

$$E = \mathcal{P}(\Sigma^*) \setminus \emptyset \qquad X \triangleright Y = X \cdot Y \qquad X \sqcup Y = X \cup Y \qquad I = \{\epsilon\}$$

*Where sequencing is pairwise concatenation of sets, $X \cdot Y = \{xy \mid x \in X \wedge y \in Y\}$.*

We write this effect quantale for a particular alphabet $\Sigma$ as $\mathsf{FinTrace}(\Sigma)$.

Intuitively, the residual should be defined whenever the dividend is some kind of prefix of all behaviors in the numerator. The formalization is more subtle, but captures this intuition:

$$X \setminus Y = \{w \in \Sigma^* \mid \forall x \in X.\, x \cdot w \in Y\} \text{ when non-empty}$$

That is, the residual $X \setminus Y$ is the set of words which, when sequenced after $X$, will produce a subset of $Y$.

Note that this is *not* the quotient of formal languages, which uses the same notation with a different meaning. The (right) *quotient* of X under Y $X \setminus_q Y$ (using the subscript $q$ to distinguish the quotient from the residual) is $\{w \in \Sigma^* \mid \exists x \in X.\, x \cdot w \in Y\}$. Sequencing this after the set $X$ yields $\{xw \mid x \in X \wedge \exists x' \in X.\, x' \cdot w \in Y\}$, which may be larger than $Y$.

The above (full) residual is in fact an operation that exists in Action Logic, a cousin of Kleene Algebra. Pratt [57] notes that the two are related; in our notation, $X \setminus Y = (X \setminus_q (Y^{-1}))^{-1}$. Since regular languages are closed under complement and quotient, this means they are also closed under residuation, and therefore there is a sub-effect-quantale $\mathsf{Reg}(\Sigma) \subseteq \mathsf{FinTrace}(\Sigma)$ which *also* has weak residuals, which can be computed using operations on finite automata (and mapped back to regular expressions for error reporting).

The induced iteration operator on this effect quantale corresponds to the Kleene star. Note, however, that while these effect quantale operations correspond conveniently to regular expressions, the effect quantale itself is not limited to regular languages: concatenation, union, and Kleene iteration are well-defined operations on *any* formal languages, including context-free, context-sensitive, or recursively enumerable languages.

### 3.3 Atomicity

Flanagan and Qadeer [21,20] proposed well-known approaches to capturing atomicity as effects. Their original proposal turns out to be a particular finite effect quantale [25,27], whose join semilattice and sequential composition are shown in Figure 2a[3] (in the bottom table, the effect in row $i$ sequenced via $\triangleright$

---

[3] They subsequently proposed an extension to conditional atomicity [20], which is also an effect quantale when combined with a data race freedom quantale.
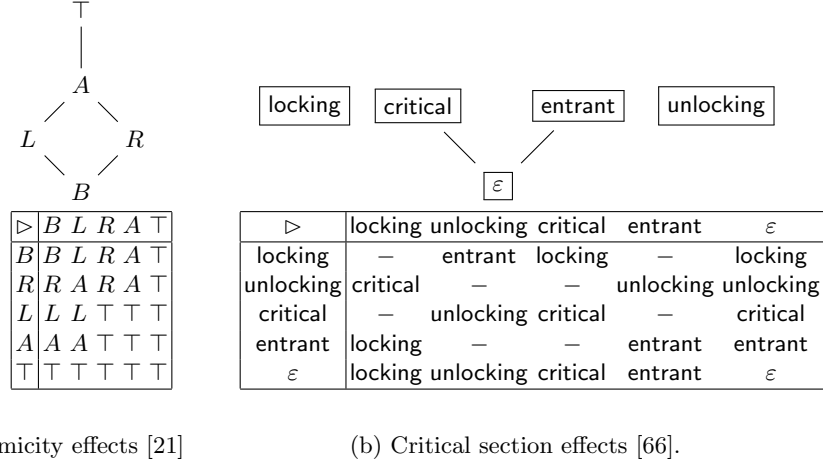
Atomicity lattice:

$$\top - A,\quad A \text{ over } L \text{ and } R,\quad L,R \text{ over } B$$

| ▷ | B | L | R | A | ⊤ |
|---|---|---|---|---|---|
| B | B | L | R | A | ⊤ |
| R | R | A | R | A | ⊤ |
| L | L | L | ⊤ | ⊤ | ⊤ |
| A | A | A | ⊤ | ⊤ | ⊤ |
| ⊤ | ⊤ | ⊤ | ⊤ | ⊤ | ⊤ |

Critical section effects: locking, critical, entrant, unlocking, $\varepsilon$

| ▷ | locking | unlocking | critical | entrant | $\varepsilon$ |
|---|---|---|---|---|---|
| locking | − | entrant | locking | − | locking |
| unlocking | critical | − | − | unlocking | unlocking |
| critical | − | unlocking | critical | − | critical |
| entrant | locking | − | − | entrant | entrant |
| $\varepsilon$ | locking | unlocking | critical | entrant | $\varepsilon$ |

(a) Atomicity effects [21]          (b) Critical section effects [66].

Fig. 2: Lattices and sequencing for atomicity and critical section (reentrancy) effects. − represents an undefined result for sequential composition.

with the effect in column $j$ is equal to the effect in cell $i, j$ of the table). The key idea is to adopt Lipton's theory of reduction [43] to label each expression with an effect capturing how it commutes with shared-memory operations in other threads: $B$ for both directions (e.g., thread-local actions), $L$ for left (i.e., earlier, such as lock releases), right ($R$, later, such as lock acquisitions), atomic $A$ (does not commute, including atomic hardware operations and already-proven-atomic critical sections), or compound $\top$ (interleaves in non-trivial ways with other threads).

Sequential composition captures Lipton's idea that any sequence of $R$ or $B$ actions, followed by at most one atomic $A$ action, then any sequence of $L$ or $B$ actions, can be grouped together as if the whole sequence occurred atomically from the perspective of another thread. We omit iteration for brevity, but the original definition of $(-)^*$ coincides with the results of a general construction on finite effect quantales as well [27].

We can define a (total) residual $\chi \setminus \chi'$ according to the classic mathematical definition: $\chi \setminus \chi' = \bigsqcup \{\chi'' \mid \chi \triangleright \chi'' \sqsubseteq \chi'\}$, which is well-defined because the join semilattice is complete. So for example, $L \setminus A$ is the greatest effect which, when sequenced *after* $L$, yields a result less than $A$ — which in this case works out to be $L$ itself.

### 3.4   Reentrancy

Tate [66] developed a maximally-general framework for sequential effect systems, and his running example was the system given in Figure 2b, which has partial joins and sequencing. (He did not describe an iteration operator, but one can be derived from the general construction of iteration operations on finite effect quantals [27].) This is an effect system motivated by tracking critical sections for a single global

$$\begin{aligned}
&\text{EFFECTS} && \chi \in Q \text{ (effect quantale)} \\
&\text{TYPES} && \tau ::= \text{bool} \mid \text{unit} \mid \tau \xrightarrow{\chi} \tau \\
&\text{EXPRESSIONS} && e ::= x \mid c \mid \lambda^\chi x : \tau.\,e \mid e@e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{while } e\ e
\end{aligned}$$

$$\boxed{\chi \mapsto \chi \parallel \Gamma \vdash e : \tau \mid \chi}$$

$$\text{T-Var } \frac{\Gamma(x) = \tau}{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash x : \tau \mid I}$$

$$\text{T-Const } \frac{}{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash c : \tau_c \mid I}$$

$$\text{T-Lambda } \frac{I \mapsto \chi \parallel \Gamma, x : \tau \vdash e : \tau' \mid \chi' \quad\quad \chi' \sqsubseteq \chi}{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash \lambda^\chi x : \tau.\,e : \tau \xrightarrow{\chi} \tau' \mid I}$$

$$\text{T-App } \frac{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e_1 : \tau' \xrightarrow{\chi_l} \tau \mid \chi_1 \qquad\qquad \chi_0 \rhd \chi_1 \mapsto \chi_m \parallel \Gamma \vdash e_2 : \tau \mid \chi_2 \quad \chi_l \setminus (\chi_2 \setminus (\chi_1 \setminus (\chi_0 \setminus \chi_m))) \text{ defined}}{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e_1@e_2 : \tau \mid \chi_1 \rhd \chi_2 \rhd \chi_l}$$

$$\text{T-If } \frac{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e_c : \text{bool} \mid \chi_c \qquad \chi_0 \rhd \chi_c \mapsto \chi_m \parallel \Gamma \vdash e_t : \tau \mid \chi_t \quad\quad \chi_0 \rhd \chi_c \mapsto \chi_m \parallel \Gamma \vdash e_f : \tau \mid \chi_f \quad (\chi_t \sqcup \chi_f) \setminus (\chi_c \setminus (\chi_0 \setminus \chi_m)) \text{ defined}}{\chi_0 \rhd \chi_m \parallel \Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_f : \tau \mid \chi_c \rhd (\chi_t \sqcup \chi_f)}$$

$$\text{T-While } \frac{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e_c : \text{bool} \mid \chi_c \quad\quad \chi_0 \rhd \chi_c \mapsto \chi_m \parallel \Gamma \vdash e_b : \tau \mid \chi_b \qquad (\chi_b \rhd \chi_c)^* \setminus (\chi_c \setminus (\chi_0 \setminus \chi_m)) \text{ defined}}{\chi_0 \mapsto \chi_m \parallel \Gamma \vdash \text{while } e_c\ e_b : \text{unit} \mid \chi_c \rhd (\chi_b \rhd \chi_c)^*}$$

Fig. 3: Type rules with and without residual checks.

resource lock which does *not* permit recursive acquisition. This turns out to *also* be a natural effect system for tracking non-reentrant code: the start of a non-reentrant operation can be given the effect locking. Notice that locking $\rhd$ locking is undefined, so attempting to reenter an operation that is non-reentrant will not type-check. This covers non-reentrant locks (as in the original example, and in implementations such as Java's `StampedLock`), but also database APIs that do not support nested transactions (starting and finishing transactions), or the evaluation API for Java's `XPath` expressions (`XPathExpression.evaluate(...)`).

### 3.5   Connecting Residuals to Type Checking

Figure 3 defines two typing judgments. $\Gamma \vdash e : \tau \mid \chi$ is a standard judgment form for effect systems, interpreted as "under variable typing assumptions $\Gamma$, expression $e$ has type $\tau$ with effect $\chi$." This judgment is readable in Figure 3 by ignoring the extensions in blue, and corresponds to a subset of the type rules Gordon [27,25] proved sound for a wide array of possible primitives and state models. In short, the judgment types expressions, using the effect quantale operators to synthesize the effect of the expression, capturing evaluation ordering with $\rhd$ and alternative paths (e.g., in T-If) with $\sqcup$. We call this judgment the *standard judgment*.

Including the text in blue, Figure 3 defines an additional judgment $\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e : \tau \mid \chi$, interpreted as "under typing assumptions $\Gamma$ expression $e$ has type $\tau$ and effect $\chi$, and moreover if $e$ is executed after effect $\chi_0$, it is still possible for the result to have effect less than $\chi_m$." Later we formalize this interpretation.

This extended judgment form performs additional checks (also in blue). These additional checks do not reject (or accept) any additional programs, but restrict how, when, and why programs are rejected.

**Lemma 1 (Conservative Extension ✿).** *If $\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e : \tau \mid \chi$, then $\Gamma \vdash e : \tau \mid \chi$.*

This lemma, and others marked with a ✿ icon, have been mechanically checked in the COQ proof assistant; the proofs are packaged with our artifact [30].

Critically, this result implies than any soundness results holding for the standard judgment are inherited by the extended judgment. Gordon [27] gives generic type safety results for a configurable framework of sequential effect systems, parameterized by primitives and choices of states. Since our standard judgment is an instantiation of that framework (for a specific choice of constants), our extended judgment is also type-safe for appropriate choices of state and primitive semantics. We do not give further consideration to type-safety in this paper, as Gordon's framework can be instantiated to yield type-safety results for all of our examples effect systems (in some cases, demonstrated in that work [27]).

Proving the other direction of the correspondence relies on weakened versions of standard residual properties:

**Lemma 2 (Residual Sequencing ✿).** *For any effects $x, y \in Q$ for a residuated effect quantale $Q$, if $x \setminus y$ is defined, then $x \rhd (x \setminus y) \sqsubseteq y$.*

**Lemma 3 (Residual Shifting ✿).** *For any effects $x, y, z \in Q$ for a residuated effect quantale $Q$, $(x \rhd y) \setminus z$ is defined if and only if $y \setminus (x \setminus z)$ is defined.*

**Lemma 4 (Antitone Residuation ✿).** *If $x \sqsubseteq y$ and $y \setminus z$ is defined, then $x \setminus z$ is defined.*

These are enough to prove that the standard typing judgment implies the extended judgment holds, for reasonable choices of $\chi_0$ and $\chi_m$. By reasonable, we mean that it is possible to "reach" $\chi_m$ by running code with effect $\chi_0$, then code with the effect the standard judgment assigns, followed by some additional effect, without exceeding a total upper bound of $\chi_m$.

**Lemma 5 (Liberal Extension ✿).** *If $\Gamma \vdash e : \tau \mid \chi$, then for any $\chi_0$ and $\chi_m$ such that $\chi \setminus (\chi_0 \setminus \chi_m)$ is defined, $\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e : \tau \mid \chi$.*

*Loop Unrolling* While not required for the proof above, readers may wonder if the desired residual is provable for finite loop unrollings, i.e., if $(\chi_b \rhd \chi_c)^n \setminus (\chi_c \setminus (\chi_0 \setminus \chi_m))$ is defined for all naturals $n$. Indeed it is: the properties of iteration imply $(\chi_b \rhd \chi_c)^n \sqsubseteq (\chi_b \rhd \chi_c)^*$, so this follows from Lemma 4.

*Completability* These results highlight that the extended judgment accepts exactly the same programs as the standard judgment, but ensures our informally-stated requirement that the extended judgment checks that given a prefix effect and target bound, the program has a valid effect in that usage context:

**Theorem 1 (Completability ✿).** *When $\chi_0 \setminus \chi_m$ is defined and $\chi_0 \mapsto \chi_m \parallel \Gamma \vdash e : \tau \mid \chi$, then $\chi \setminus (\chi_0 \setminus \chi_m)$ is defined.*

This result highlights a subtlety of the nested residual checks in several rules. While at a glance it may appear to lump many checks into one since the residuals checked for existence are larger nested residuals, these checks actually verify only one residual definition beyond what is guaranteed by the subexpressions' typing results. Consider as an example T-App. If the antecedent subexpression typings hold, then by Completability $(\chi_1 \setminus (\chi_0 \setminus \chi_m))$ is defined and $\chi_2 \setminus (\chi_1 \setminus (\chi_0 \setminus \chi_m))$ is defined. So T-App is truly checking only the residual with respect to the latent effect of the invoked function. Likewise, T-If and T-While are checking only the residual with respect to the respective constructs, with residuals for the subexpression effects already guaranteed by the antecedents. Later this allows us to ensure that algorithmically, only one residual check is required per source construct, and that if it would fail due to a subexpression rather than the construct being checked, that error would have already been reported when checking the subexpressions individually.

The combination of Theorem 1 and Lemma 5 guarantees that if an expression is well-typed under standard typing, but not under extended typing for a given $\chi_0$ and $\chi_m$ with $\chi_0 \setminus \chi_m$ defined, it is because the subexpressions are arranged in a way that is incompatible with the effect context $\chi_0 \mapsto \chi_m$. And because the residual checks in each rule fail (the residual becomes undefined) as soon as any execution prefix's residual is undefined, if the extended judgment rejects a term accepted by the standard judgment, the specific (nested) residual check pinpoints the inadmissible execution prefix for that context.

### 3.6   Limitations

For the residual to localize errors effectively, it must be the case that some residuals are undefined, since if all residuals exist, then there is no basis for issuing the early errors we seek. A trivial example is the effect quantale given by a group with only the reflexive ordering. In such an effect quantale, it is always possible to start with a prefix effect $X$ and reach a final effect $Y$ by sequencing $X^{-1}Y$ after $X$, so no residual check will fail — only the final subeffect check in T-Lam. While such effect quantales exist mathematically, we are unaware of any such quantales actually being used as effect systems, as conceptually, they are at odds with the typical goal of an effect system, which is to build a sound summary of (selected) program behavior and reject certain subsets.

Kleene Algebras [40] are iterable effect quantales [27] which are typically used in program analysis, and typically include a least element in lattice order which corresponds to "no behavior" (e.g., an empty set or empty relation). Since most programs are expected to have some kind of behavior, one could modify a Kleene Algebra to be partial (i.e., removing the least element from its domain) and obtain an effect quantale with meaningful notions of residuals. However, even then there may be "too many" residuals to be useful. For example, in a Kleene Algebra of binary relations (pre- and post-conditions) or of transition

functions (e.g., modeling abstract interpretation as a Kleene Algebra [39]), the domain is large enough to admit arbitrary transitions: for any prefix behavior with precondition $P$ as a set of states and ultimate goal postcondition $Q$ as a set of states, the relation $P \times Q$ is non-empty as long as both $P$ and $Q$ are non-empty. So applying our technique to domains like these notions of extensional correctness would require refining the set of transition functions or relations considered, in order to make the residual undefined when no action contained in the actual program could lead to a desired execution. This is less problematic for intensional specifications such as the language-theoretic approaches, because as soon as some execution path commits to a behavior prefix not in the target specification, the residual is undefined.

## 4   Algorithms

To turn our insights into a tool, we require a type-checking algorithm. Figure 4 shows the core pieces of a type-checking algorithm that accepts exactly the programs well-typed under the extended typing judgment of Figure 3.[4] We assume that `earlycheck` accepts the contextual and goal effects, type environment, and expression as inputs, and returns a result (a supertype of successful typing results, unbound variable errors, type errors, and effect errors).

The key insight of the residual checking is apparent in the case for applications. This case first type-checks the function position. If that subterm is already problematic then the underlying error is returned. Otherwise, the algorithm continues to ensure the function subterm has a function type, then type-checks the argument position *with a modified progress effect* reflecting that it reduces after the function position (following T-App). If both subterms typecheck (in appropriate contexts), and the argument position is of an appropriate type for the function, then the algorithm checks that the residual corresponding to the final check in T-App is defined, returning success in that case. Note that while there is only one residual check in the application case here, errors are still reported as early as possible in program order. In particular:

- If the function subterm's effect could not be sequenced after $\chi_0$, or the result would not leave the appropriate residual defined, then type-checking of that *subterm* would fail.
- Similarly, if the argument subterm's effect could not be sequenced after $\chi_0$ and the function subterm's effect, type-checking of the argument would fail (note that the function's effect is passed into type-checking for the argument).
- The only residual not guaranteed to be defined is the final step of the residual checking, involving the function's *latent* effect, whose interaction with the evaluation context is not implied by the function and argument subterm effects.

The cases for conditionals and loops are similar, checking subexpressions in program order, with only one new explicit residual check in each case.

---

[4] The figure uses Java 17's extended switch statements, plus a few notational liberties for operations on effect quantale elements.

```
Result earlycheck(Q χ₀, Q χₘ, env Γ, expr e) {
  switch (e) {
    case Var x:
      Type t = lookup(Γ, x);
      return (t != null) ? new Result(t, I) : new Unbound(x);
    case Const c: return new Success(constType(c), I);
    case Lambda lam:
      Result r =
        earlycheck(I, lam.declEffect(), Γ.with(lam.var(), lam.argty()), lam.body());
      if (!r.isSuccess()) return r;
      Success s = (Success) r;
      return new Success(new FunType(lam.argty(), lam.declEffect(), s.type), I);
    case App app:
      Result resa = earlycheck(χ₀, χₘ, Γ, app.fun);
      if (!resa.isSuccess()) return resa;
      Success sa = (Success) resa;
      /* We know sa.effect\(χ₀\χₘ) is defined */
      if (!sa.type.isFun()) return new BadType(app.fun);
      Result resb =  earlycheck(χ₀▷sa.effect, χₘ, Γ, app.arg);
      if (!resb.isSuccess()) return resb;
      Success sb = (Success) resb;
      /* sb.effect\(sa.effect\(χ₀\χₘ)) defined */
      if (!sa.type.arg.equals(sb.type)) return new BadType(app.arg);
      boolean goodEffect = hasResidual(sa.type.asFunc().latentEffect(),
                                  (sb.effect \ (sa.effect \ (χ₀ \ χₘ))));
      if (goodEffect) {
        return new Success(sa.type.result,
                    sa.effect ▷ sb.effect ▷ sa.type.asFunc().latentEffect());
      } else {
        return new BadEffect(app);
      }
    case If i: ...
    case While w: ...
  }
}
```

Fig. 4: Algorithm for early effect errors.

## 5   Implementation

We have implemented this approach in a prototype extension of the Checker Framework [53,13] to support sequential effect systems. Currently we support the fragment of Java corresponding to a core object-oriented language: classes, methods (including checking a method override's effect is less than the original's), conditionals, while loops, calls, and switch statements. Exceptions are supported through a variant of Gordon's work on tagged delimited continuations [26], described in more detail in our full technical report [31].

```java
public abstract class EffectQuantale<Q> {
  public abstract Q LUB(Q l, Q r);
  public abstract Q seq(Q l, Q r);
  public abstract Q unit();
  public abstract Q iter(Q x);
  public abstract Q residual(Q sofar, Q target);
  public boolean LE(Q l, Q r) { return LUB(l, r).equals(r); }
  public boolean isCommutative() { return false; }
  public abstract ArrayList<Class<? extends Annotation>> getValidEffects();
}
```

Fig. 5: Framework interface to effect quantales with effects represented by type `Q`.

The framework extension is parameterized by a choice of effect quantale, represented by the abstract class in Figure 5, which is parameterized by the representation type `Q` for a given system's effects. It contains operations for $\sqcup$ (`LUB`), $\rhd$ (`seq`), unit, $-^*$ (`iter`), and residual checks (`residual`). Partiality is modeled by returning `null`. A default implementation of $\sqsubseteq$ (`LE`) is provided but can be overridden with more efficient implementations. `getValidEffects()` produces a list of Java annotation types the framework should recognize as being part of this effect quantale.

`isCommutative` indicates whether the effect quantale is commutative, which the framework uses to recover exhaustive error checking for such systems. In general, when the effect of an expression is $\alpha \rhd \beta \rhd \gamma$ and the residual $(\alpha \rhd \beta) \setminus \delta$ is undefined, the framework stops issuing errors about the rest of execution on the same path through the current method, because all residuals with more complete body effects (e.g., $(\alpha \rhd \beta \rhd \gamma) \setminus \delta$) will also be undefined, but not necessarily because of problems with the extensions (i.e., $\gamma$ may be fine if the problem is $\beta$, in which case further errors would be redundant). However, in the case of a commutative $\rhd$, the framework can exploit commutativity to give additional error messages. For example, if $(\alpha \rhd \beta) \setminus \delta$ is undefined as before, then $(\alpha \rhd \beta \rhd \gamma) \setminus \delta$ will also be undefined, but $\alpha \rhd \beta \rhd \gamma = \alpha \rhd \gamma \rhd \beta$, and it is possible that $(\alpha \rhd \gamma) \setminus \delta$ may be defined or undefined, independent of the residual involving $\beta$. This is precisely why the standard approach for join-semilattice effect systems of checking individual subexpressions' effects against a bound works and gives all appropriate errors: in our setting, the residual is defined as the bound itself as long as the effect so far is less than the bound, which via commutativity extends to the join of any non-empty subset of body effects being less than the bound. Note however that our approach is general to any commutative effect quantale, including systems like must-effect analysis [47], not just join semilattices.

General checking logic in the Checker Framework (as in the rest of the Java compiler) uses a visitor to traverse ASTs, rather than recursive traversals in Figure 4. The Java compiler provides the type environment as ambient state in this setting, and the implementation also maintains the current $\chi_m$ and $\chi_0$ as visitor state — $\chi_0$ is maintained as a stack of subexpression effects which can be rewound to consider alternate paths (e.g., different branches of a conditional).

But the core algorithm is as demonstrated in the application case of Figure 4, with a single explicit residual check in each case.

We do make two kinds of extensions to the logic. First, in addition to other varieties of loops (which are straightforward adaptations of T-WHILE), we handle additional language constructs present in Java: our handling of exceptions, breaks, and early returns follows an extension of Gordon's work on effect systems for tagged delimited continuations [26]. Gordon defines a transformation of an arbitrary effect quantale that is ignorant of non-local control flow to one that works with tagged delimited continuations. From this, typing rules for checked exceptions, break statements, and early returns can be derived, which we implement. We also extend this construction with residuals, and it is this extension which requires our shift to weak residuals: intuitively, Gordon's construction tracks sets of possible effects for each execution path (normal execution, for each thrown exception, etc.). In general, the *greatest* possible residual for this construction may require modeling an infinite set, while our implementation relies on finite sets. Further details of our handling and residual are given in our extended technical report [31].

Second, when a type error is encountered, the algorithm does not immediately stop. It will immediately report the error, but then sets a flag indicating an error has been found on the current path, suppressing further error reporting. Then traversal continues in order to visit subexpressions that correspond to checking different method bodies (i.e., lambda expressions and anonymous inner classes) and report errors there, which are independent of errors in the surrounding code. When checking conditionals, if an error is encountered in one branch the algorithm resets the flag for an error on the current path and checks the other branch as if it were the only branch (including visiting code later in program order than the whole conditional construct). This permits the implementation to report additional independent errors, rather than allowing errors in one branch to shadow errors in the other.

Our extension inherits the Checker Framework's existing robust support for subtyping, and type generics. While not used in any of our evaluations, the base implementation of the effect visitor extends the base implementation used for the Checker Framework's focus on type qualifiers [22], so effect systems can in principle make use of type qualifiers to determine effects.

Effects are declared as Java annotations targeting method nodes. This unfortunately requires a bit of boilerplate: each effect requires 10 lines of code, but 8 are identical across all declarations (import statements and meta-annotations for the Java compiler to allow them on method declarations and persist them in bytecode), with the remaining lines being the package declaration and one line for naming the actual annotation.

*Performance* The execution time of these checkers is dependent primarily on two factors. The first is the underlying effect quantale: if computing sequencing, joins, iteration, and residuals for the underlying effect quantale is particularly slow, this will slow the whole framework. The atomicity and reentrancy effect systems we have implemented (Section 6) both have very fast basic effect quantale operations.

The second is the cost of working with the control effect transformation of the underlying effect system to handle exceptions, breaks, and non-local returns without individual effect systems needing to address them (see [31]). In the common case (no non-local control flow) an effect represented as an object with an underlying effect representation, and two null pointers, so the operations have very little additional cost over ignoring non-local control entirely. In code that contains non-local control flow or calls methods with checked exceptions, a prefix effect characterizing behavior up to each break, non-local return, throw, and checked exception mentioned in the signatures of called methods. Composing such effects may trigger at most $n$ additional calls to the underlying effect quantale when each effect is tracking at most $n$ non-local behaviors, so the costs do not grow significantly with code complexity.

Critically, since we analyze a single method at a time, even when those sets become large their performance impact is confined to the current method being checked. The only way for complexity of one method to influence the cost of checking another is via method annotations that expose latent effects for a variety of different thrown exceptions in addition to the non-exceptional method body effect. In general the number of such annotations that must appear is dependent not only on the program being analyzed, but also on the specific effect system in use.

In practice, the performance of our implementation, for the inexpensive effect quantales described in the next section, is on par with other existing pluggable type systems in the Checker Framework.

## 6  Evaluation

The hypothesis underlying this work is that residuals offer a way to localize sequential effect system errors to the earliest program location in program order where a mistake can be recognized, and that this useful precision for developers.

We have implemented two sequential effect systems in this framework to evaluate whether the error locations reported are accurate, which we approach from two angles. First, we reproduce part of Flanagan and Qadeer's evaluation of their atomicity effect system [20], where they found atomicity errors in the then-current JDK (which have since been fixed). The original evaluation simply described the errors as being found as a result of their analysis, with the implication that their tool used global reporting and thus all location of the error was manually driven. Second, we evaluate the accuracy of error reporting for the reentrancy effect system applied to non-reentrant database transactions. This is a common situation across Spring Hibernate, JDBC, and other database systems, and is a situation with non-trivial interactions with exceptions. We find that the residual-based error locations are both predictable and accurate. We evaluate these two systems because they cover both total and partial effect quantales, and because they are simple enough (both are finite with 5 effects each) that we believe we can evaluate the residual-based locations of errors without becoming entangled in deeper questions of how complex error messages are presented, which we believe is important future work for systems with more complex effects,

```
@Atomic
public boolean contentEquals(StringBuffer sb) {
    if (count != sb.length()) // length() is atomic
        return false;
    char v1[] = value;
    char v2[] = sb.getValue(); // <-- Error: second atomic operation
    int i = offset;
    int j = 0;
    int n = count;
    while (n-- != 0) {
        if (v1[i++] != v2[j++])
            return false;
    }
    return true;
}
```

Fig. 6: Excerpt from JDK1.4 `String` implementation.

e.g. Section 3.2) where the relationship between the residual's existence and the effects a programmer specifies, are more complex.

A snapshot of our implementation and these case studies is available [30].

### 6.1   Reproducing Atomicity Errors

We have implemented the earlier of Flanagan and Qadeer's systems for static checking of atomicity via effects [21] (introduced in Section 3.3), in 201LOC, 50 of which were for the 5 effect declarations. A full reimplementation would require integration with a data race freedom system [27] (since data races are non-atomic, while well-synchronized memory accesses are both-movers); this version assumes data race freedom. We have run our atomicity checker with residual-based error reporting on some of the JDK1.4 Java classes reported in Flanagan and Qadeer's evaluation to have errors, to check error location accuracy.

The most prominent example in their evaluation was an atomicity violation in the `StringBuffer` code in Figure 1. The bug in the code is that while the code synchronizes on (locks) the receiver `this`, it performs *two* atomic actions in the body (calls two atomic methods on the argument, which is not locked), even though it is supposed to be atomic. Our prototype reports the second atomic operation, which is the first subexpression in the body that makes it impossible for the remainder of the method to have an overall atomic effect.

Flanagan and Qadeer report a similar bug in `java.lang.String` (of JDK 1.4), shown in Figure 6. There the same `StringBuffer` methods are involved. Again our technique indicates the exact point in the method beyond which an overall method body effect consistent with the annotation is impossible. In this case, it saves the developer the trouble of looking at most of the method code.

Flanagan and Qadeer also analyzed other parts of the JDK 1.4, but we have had difficulty getting other classes from their evaluation to be accepted with only minor modifications by the modern Java compiler the Checker Framework extends.

```
@Entrant
@ThrownEffect(exception = TxException.class, behavior=Entrant.class)
public void docExample(SessionFactory factory) throws TxException {
  Session sess = factory.openSession();
  Transaction tx = null;
  try {
    tx = sess.beginTransaction();
  } catch (TxException e) {
    sess.close();
    throw e;
  }
  try {
    doWork(tx);
    tx.commit();
  } catch (TxException e) {
    tx.rollback();
    throw e;
  } finally {
    sess.close();
  }
}
```

Fig. 7: Slight refactoring of example from Hibernate documentation.

### 6.2 Reentrancy

We have also implemented Tate's system for reentrancy checking [66] (introduced in Section 3.4). In this system, the critical effect describes code which is safe to use inside a critical section, but which may not begin another (nested) critical section or end the current (presumed) critical section. For our evaluation, rather than focusing on non-reentrant locks (which are little-used in Java), we have focused instead on database transactions, as some database systems (notably Hibernate) do not support nested transactions at all, while general-purpose database interfaces like JDBC leave the behavior of nested transactions up to the particular backend chosen (making the use of nested transactions clearly wrong for some backends, and more generally undefined behavior).

For our case study we focus on a variant of JBoss Hibernate's transaction API.[5] Consider the code in Figure 7 from Hibernate's documentation.[6] Despite

---

[5] This is a variant of the current API with checked exceptions. We must currently use a variant because the Checker Framework's support for stub files (a means to externally annotate compiled JAR files) does not work with multiple checkers "owning" the @ThrownEffect annotation. While this certainly affects the real-world applicability of our checker as a tool, it does not impact our evaluation of error reporting accuracy against an extracted copy of the API, and our focus is evaluation of the residual-based error reporting.

[6] https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/Session.html#beginTransaction()

its small size, the main method `docExample` (correctly) handles several significant subtleties. The main transaction itself is in a `try-catch` block, as the session and transaction methods may throw an exception. The `doWork()` method, a factored out transaction body, is marked `@Critical`. The commit method is annotated as

```
@Unlocking
@ThrownEffect(exception=TxException.class, behavior=Basic.class)
public void commit();
```

indicating that if it succeeds it behaves as if unlocking (i.e., finishing the transaction), while if it throws a `SQLException` the transaction remains open. This ensures that if the body throws an exception, the catch block is checked assuming the code has not yet finished the transaction, requiring the rollback attempt. Because rollback is typically a last-resort fallback, it is annotated as

```
@Unlocking
@ThrownEffect(exception=TxException.class, behavior=Unlocking.class)
public void rollback();
```

since if it fails, the connection is almost certainly broken, and the database will automatically rollback the transaction after a timeout.

Unlike the atomicity system, some sequential compositions in the reentrancy effect system are already undefined (recall Figure 2b), and would be immediately and locally rejected even without residual-based error detection, which is focused on cases where composition is defined but has already committed the program being analyzed to a course already known to be incompatible with its top-level effect specification. So, for example, attempting to start an additional nested transaction would have effect `@Locking`▷`@Locking`, which is undefined and would be rejected without our extensions. The additional cases which are rejected earlier due to residual-based error checking are related to the rows of Figure 2b's ▷ table lacking certain operations: notice that each non-unit row of the table has either locking and entrant results, or unlocking and critical results. The former rows (for locking ▷ − and entrant ▷ −) correspond to cases where the code has already committed to being code that must start running *not* inside a transaction, so have no residual with respect to a method-level annotation that the method's code should be able to start inside a transaction (i.e., have the critical or unlocking) effect.

The additional errors thus manifest in refactoring of the example code's actual work into `doWork`. In large projects, it is easy to lose track of the intended execution context of a method [28], sometimes resulting in developers incorrectly assuming they may need to construct some of that context themselves. While directly starting a nested transaction in the same syntactic scope as another transaction start would be undefined and therefore reported precisely even without our extension (one of the original, informal arguments in favor of effect quantales being partial [27]), factoring the body of the transaction out into this helper method requires annotating the transaction body method with an effect. If it were annotated with `@Entrant`, the call site (specifically) would be rejected as undefined (since `@Locking`▷`@Entrant` is undefined). Annotating it as `@Critical` (as we do)

the call site is accepted, but the factored-out method would be rejected, raising the question of where the error would be reported. Our technique reports the start of the nested transaction as the error location even if it is the first line of code:

```
@Critical
@ThrownEffect(exception=TxException.class, behavior=Critical.class)
public void doWork(Transaction tx) throws TxException {
  ...
  tx.begin(); // <-- error reported
  ...
}
```

Thus also with an effect quantale with partially-defined compositions, residual-based error checking yields additional precise error locations.

## 7  Related Work

The most closely-related work to ours is that on implementations of sequential effect systems. The implementations we know of for concrete sequential effect systems [63,20] do not have error handling described in the corresponding publications, but are formalized in the standard way which corresponds to the all-at-once method behavior check. It is possible that these systems implemented some kind of eager error reporting in the tools themselves, but the sources are no longer available and in any case these would be optimizations for specific effect systems. Our results establish a profitable eager error reporting strategy for *many* sequential effect systems expressible as effect quantales (it is not necessarily the case that all effect quantales have a residual operator as we propose, but all those described as effect quantales in the literature [27] have residual operators).

There have also been a number of generalized implementations of sequential effect system frameworks. Hicks et al. [34] implement a general elaboration to polymonadic effects, which are equivalent to Tate's producers [66]. They use a constraint-based approach to determine which specific monad each expression should be in (and therefore the effect of each expression). Orchard and Petricek [52] and Bracker and Nilsson [8] implement an embedding of graded monads into Haskell, using typeclass constraints to define the composition and lifting operations. Because this is constraint-based, effect errors will be issued at an arbitrary program point corresponding to a failed constraint, which as in general type inference may be not directly related to the actual mistake in the program. Bracker and Nilsson [9,10] later defined *supermonads*, which generalize many monadic computation types by generalizing to an arbitrary number of parameters to a monadic type (vs. 1 for indexed monads [70], 2 for parameterized monads [3]), and can be used to express polymonads. They also added specialized support for supermonad constraints to Haskell's type inference. Ultimately, because the constraints are still solved in an arbitrary solver-selected order, this suffers from the same problems with unpredictable error locations that exist in the normal Haskell implementations, where errors may be issued in unproblematic program

locations, and program changes unrelated to the error may change where an error is reported by affecting constraint solving order. Our approach yields predictable error locations with some guaranteed relevance to actual program errors.

More broadly, there is a wealth of work on better localizing type errors in constraint-based type inference. Many techniques have been applied with a wide variety of trade-offs. Most of this involves searching for a minimal program or type repair that results in inference succeeding [73,55,54,44,12], and reports the location whose term or type was assumed to change or whose type constraint was removed as the most likely error location (in general a program with a general type error may have many incompatible constraints, so the smallest number of changes or removals that fixes the most incompatibilities is likely a source). These approaches are all quite sensible, though both their approaches and setting differs significantly from ours. None of this work on localizing type inference errors treats effects (notably, none of it targets a language in which monads are used to encode effects, though in principle these techniques could be applied to Haskell and thus the Haskell embeddings of effects above). The assumptions available to us for our work are also much stronger in some ways than what is available to the general type inference localization problem. Because sequential effects are so closely coupled with evaluation order, there is a semantically-meaningful notion of best error location, while in general type inference the earliest inconsistency in program order may not be meaningfully related to the actual error location (hence the common practice there of exploring formalizations of the intuitive notions of "minimum changes to fix"). In addition, our effects have far more structure than typical type inference problems, because compared to general bags of constraints as in HM(X) [51] or extensions thereof for object-orientation [56,69], effect quantales afford a convenient algebraic characterization of an operation useful for error location: the residual. As a result, our work is the first which takes an *algebraic* approach to localizing effect errors.

## 8    Conclusions

We have proposed the first algebraic approach to localizing errors in sequential (flow-sensitive) effect systems, by exploiting the notion of a partial residual. This approach is guaranteed to give more precise error locations than the method-global (and therefore highly imprecise) or constraint-based (and therefore unpredictable) techniques used in all prior sequential effect system implementations, locations which are moreover guaranteed to have relevance to the actual program mistake. We have implemented our technique for Java in a fork of the open source Checker Framework, and shown that our technique gives specific meaningful error messages for previously studied bugs.

# References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst. **28**(2) (2006)
2. Amtoft, T., Nielson, F., Nielson, H.R.: Type and Effect Systems: Behaviours for Concurrency. Imperial College Press, London, UK (1999)
3. Atkey, R.: Parameterised Notions of Computation. Journal of Functional Programming **19**, 335–376 (July 2009)
4. Bao, Y., Wei, G., Bračevac, O., juan, Y., He, Q., Rompf, T.: Reachability types: Tracking aliasing and separation in higher-order functional programs. Proceedings of the ACM on Programming Languages **5**(OOPSLA) (2021)
5. Birkhoff, G.: Lattice theory, Colloquium Publications, vol. 25. American Mathematical Soc. (1940), third edition, eighth printing with corrections, 1995.
6. Bocchino, Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A Type and Effect System for Deterministic Parallel Java. In: OOPSLA (2009). `https://doi.org/10.1145/1640089.1640097`
7. Boyapati, C., Lee, R., Rinard, M.: Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In: OOPSLA (2002)
8. Bracker, J., Nilsson, H.: Polymonad programming in haskell. In: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages. IFL '15, Association for Computing Machinery, New York, NY, USA (2015). `https://doi.org/10.1145/2897336.2897340`, `https://doi.org/10.1145/2897336.2897340`
9. Bracker, J., Nilsson, H.: Supermonads: one notion to bind them all. In: Proceedings of the 9th International Symposium on Haskell. pp. 158–169 (2016)
10. Bracker, J., Nilsson, H.: Supermonads and superapplicatives. Journal of Functional Programming p. 103 (2018)
11. Chen, S., Erwig, M.: Counter-factual typing for debugging type errors. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 583–594 (2014)
12. Chen, S., Erwig, M.: Systematic identification and communication of type errors. Journal of Functional Programming **28** (2018)
13. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.: Building and using pluggable type-checkers. In: ICSE (2011)
14. Dilworth, R.P.: Non-commutative residuated lattices. Transactions of the American Mathematical Society **46**(3), 426–444 (1939)
15. Flanagan, C., Abadi, M.: Object Types against Races. In: CONCUR (1999)
16. Flanagan, C., Abadi, M.: Types for Safe Locking. In: ESOP (1999)
17. Flanagan, C., Freund, S.N.: Type-Based Race Detection for Java. In: PLDI (2000). `https://doi.org/10.1145/349299.349328`
18. Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation. pp. 47–58 (2005)
19. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for atomicity: Static checking and inference for java. ACM Transactions on Programming Languages and Systems (TOPLAS) **30**(4), 1–53 (2008)
20. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI (2003)
21. Flanagan, C., Qadeer, S.: Types for atomicity. In: TLDI (2003)

22. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. pp. 192–203. PLDI '99, ACM (1999). `https://doi.org/10.1145/301618.301665`

23. Galatos, N., Jipsen, P., Kowalski, T., Ono, H.: Residuated lattices: an algebraic glimpse at substructural logics, Studies in Logic and the Foundations of Mathematics, vol. 151. Elsevier (2007)

24. Gifford, D.K., Lucassen, J.M.: Integrating Functional and Imperative Programming. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. LFP '86 (1986)

25. Gordon, C.S.: A Generic Approach to Flow-Sensitive Polymorphic Effects. In: Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP'17). Barcelona, Spain (June 2017). `https://doi.org/10.4230/LIPIcs.ECOOP.2017.13`

26. Gordon, C.S.: Lifting Sequential Effects to Control Operators. In: Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP'20). Berlin, Germany (July 2020). `https://doi.org/10.4230/LIPIcs.ECOOP.2020.23`

27. Gordon, C.S.: Polymorphic Iterable Sequential Effect Systems. ACM Transactions on Programming Languages and Systems (TOPLAS) **43**(1) (April 2021). `https://doi.org/10.1145/3450272`

28. Gordon, C.S., Dietl, W., Ernst, M.D., Grossman, D.: JavaUI: Effects for Controlling UI Object Access. In: Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13). Montpellier, France (July 2013). `https://doi.org/10.1007/978-3-642-39038-8_8`

29. Gordon, C.S., Ernst, M.D., Grossman, D.: Static Lock Capabilities for Deadlock Freedom. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'12). Philadelphia, PA, USA (January 2012). `https://doi.org/10.1145/2103786.2103796`

30. Gordon, C.S., Yun, C.: Artifact for error localization for sequential effect systems (August 2023). `https://doi.org/10.6084/m9.figshare.23822877`

31. Gordon, C.S., Yun, C.: Error localization for sequential effect systems (extended version). No. arXiv cs.PL 2307.15777 (July 2023). `https://doi.org/10.48550/arXiv.2307.15777`

32. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification: Java SE 8 Edition. Pearson Education (2014)

33. Hassan, M., Urban, C., Eilers, M., Müller, P.: Maxsmt-based type inference for python 3. In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30. pp. 12–19. Springer (2018)

34. Hicks, M., Bierman, G., Guts, N., Leijen, D., Swamy, N.: Polymonadic programming. Electronic Proceedings in Theoretical Computer Science **153**, 79–99 (Jun 2014). `https://doi.org/10.4204/eptcs.153.7`, `http://dx.doi.org/10.4204/EPTCS.153.7`

35. Ivašković, A., Mycroft, A.: A graded monad for deadlock-free concurrency (functional pearl). In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell. pp. 17–30 (2020)

36. Ivašković, A., Mycroft, A., Orchard, D.: Data-flow analyses as effects and graded monads. In: 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020). vol. 167. Dagstuhl (2020)

37. Katsumata, S.y.: Parametric effect monads and semantics of effect systems. In: POPL (2014)

38. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: CSL/LICS (2014)
39. Kot, L., Kozen, D.: Second-order abstract interpretation via kleene algebra. Tech. rep., Cornell University Ithaca (2004)
40. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems (TOPLAS) **19**(3), 427–443 (1997). https://doi.org/10.1145/256167.256195
41. Lambek, J.: The mathematics of sentence structure. The American Mathematical Monthly **65**(3), 154–170 (1958)
42. Lerner, B.S., Flower, M., Grossman, D., Chambers, C.: Searching for type-error messages. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 425–434 (2007)
43. Lipton, R.J.: Reduction: A Method of Proving Properties of Parallel Programs. Communications of the ACM **18**(12), 717–721 (Dec 1975). https://doi.org/10.1145/361227.361234
44. Loncaric, C., Chandra, S., Schlesinger, C., Sridharan, M.: A practical framework for type inference error explanation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 781–799 (2016)
45. Long, Y., Liu, Y.D., Rajan, H.: Intensional Effect Polymorphism. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming (ECOOP 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 37, pp. 346–370. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.346, http://drops.dagstuhl.de/opus/volltexte/2015/5221
46. Lucassen, J.M., Gifford, D.K.: Polymorphic Effect Systems. In: POPL (1988)
47. Mycroft, A., Orchard, D., Petricek, T.: Effect systems revisited — control-flow algebra and semantics. In: Semantics, Logics, and Calculi. Springer (2016)
48. Neamtiu, I., Hicks, M., Foster, J.S., Pratikakis, P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In: POPL. pp. 37–49 (2008)
49. Nielson, F., Nielson, H.R.: From cml to process algebras. In: CONCUR (1993)
50. Nielson, H.R., Nielson, F.: Communication analysis for concurrent ml. In: ML with Concurrency: Design, Analysis, Implementation, and Application, pp. 185–235. Springer (1997)
51. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and practice of object systems **5**(1),  35 (1999)
52. Orchard, D., Petricek, T.: Embedding effect systems in haskell. In: Proceedings of the 2014 ACM SIGPLAN symposium on Haskell. pp. 13–24 (2014)
53. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: ISSTA (2008)
54. Pavlinovic, Z., King, T., Wies, T.: Finding minimum type error sources. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 525–542 (2014)
55. Pavlinovic, Z., King, T., Wies, T.: Practical smt-based type error localization. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 412–423 (2015)
56. Pottier, F.: A Framework for Type Inference with Subtyping. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98). pp. 228–238 (Sep 1998). https://doi.org/10.1145/291251.289448

57. Pratt, V.: Action logic and pure induction. In: European Workshop on Logics in Artificial Intelligence. pp. 97–120. Springer (1990). `https://doi.org/10.1007/BFb0018436`

58. Rytz, L., Odersky, M., Haller, P.: Lightweight Polymorphic Effects. In: European Conference on Object-Oriented Programming (ECOOP 2012) (2012). `https://doi.org/10.1007/978-3-642-31057-7_13`

59. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analysis of data races and atomicity. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 83–94 (2005)

60. Bañados Schwerter, F., Garcia, R., Tanter, E.: A theory of gradual effect systems. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 283–295. ICFP '14, ACM (2014). `https://doi.org/10.1145/2628136.2628149`

61. Skalka, C.: Types and trace effects for object orientation. Higher-Order and Symbolic Computation **21**(3) (2008)

62. Skalka, C., Darais, D., Jaeger, T., Capobianco, F.: Types and abstract interpretation for authorization hook advice. In: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF). pp. 139–152. IEEE (2020)

63. Skalka, C., Smith, S., Van Horn, D.: Types and trace effects of higher order programs. Journal of Functional Programming **18**(2) (2008)

64. Suenaga, K.: Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In: APLAS (2008)

65. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of functional programming **2**(03), 245–271 (1992). `https://doi.org/10.1017/S0956796800000393`

66. Tate, R.: The sequential semantics of producer effect systems. In: POPL (2013)

67. Tofte, M., Talpin, J.P.: Region-based memory management. Information and computation **132**(2), 109–176 (1997)

68. Toro, M., Tanter, E.: Customizable gradual polymorphic effects for scala. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 935–953. OOPSLA 2015, ACM (2015). `https://doi.org/10.1145/2814270.2814315`

69. Trifonov, V., Smith, S.F.: Subtyping Constrained Types. In: Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings. pp. 349–365 (1996). `https://doi.org/10.1007/3-540-61739-6_52`

70. Wadler, P., Thiemann, P.: The Marriage of Effects and Monads. Transactions on Computational Logic (TOCL) **4**, 1–32 (2003)

71. Ward, M., Dilworth, R.P.: Residuated lattices. Transactions of the American Mathematical Society **45**(3), 335–354 (1939)

72. Yi, J., Disney, T., Freund, S.N., Flanagan, C.: Cooperative types for controlling thread interference in java. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 232–242 (2012)

73. Zhang, D., Myers, A.C.: Toward general diagnosis of static errors. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 569–581 (2014)