First-Class Effect Reflection for Effect-Guided Programming

Yuheng Long$^{\alpha}$, Yu David Liu$^{\beta}$, and Hridesh Rajan$^{\gamma}$

$^{\alpha,\gamma}$Iowa State University, USA $^{\beta}$SUNY Binghamton, USA

$^{\alpha,\gamma}${csgzlong,hridesh}@iastate.edu $^{\beta}$davidl@cs.binghamton.edu

Abstract

This paper introduces a novel type-and-effect calculus, *first-class effects*, where the computational effect of an expression can be programmatically reflected, passed around as values, and analyzed at run time. A broad range of designs "hardcoded" in existing effect-guided analyses — from thread scheduling, version-consistent software updating, to data zeroing — can be naturally supported through the programming abstractions. The core technical development is a type system with a number of features, including a hybrid type system that integrates static and dynamic effect analyses, a refinement type system to verify application-specific effect management properties, a double-bounded type system that computes both over-approximation of effects and their under-approximation. We introduce and establish a notion of soundness called *trace consistency*, defined in terms of how the effect and trace correspond. The property sheds foundational insight on "good" first-class effect programming.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures; F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords first-class effect, type system, hybrid typing

1. Introduction

Type-and-effect systems, either purely static [37, 43, 52], dynamic [4, 29, 46] or hybrid [2, 35], have proven to be useful for program construction, reasoning, and verification. In existing approaches, the logic of accessing effects and making decisions over them is defined by the language designer, and supported by the compiler or the runtime system. The end-user programmer is generally a consumer of the "hardcoded" logic for effect management.

Our work is motivated by two fundamental questions. First, are there benefits of empowering programmers with application-specific effect management? Second, is there a principled design for the effect management, so that programmers are endowed with powerful abstractions while in the meantime provided with strong correctness guarantees?

In this paper, we develop *first-class effects*, a novel type-and-effect system where the effects of program expressions are available as first-class values to programmers. The lifecycle of effect management over program expressions becomes part of the program itself. The resulting calculus, $\lambda_{\mathtt{fe}}$, is endowed with powerful programming abstractions:

[Effect Reflection] Programmers can *query* the effect of any expression $\mathtt{e}$ through the primitive **query** $\mathtt{e}$. The result is a first-class value we call *effect closure*, which contains expression $\mathtt{e}$ and $\mathtt{e}$'s effects, computed by $\lambda_{\mathtt{fe}}$'s type system, in the form of memory region accesses. For convenience, we call $\mathtt{e}$ the *passenger* expression of the closure.

[Effect Inspection] The memory access details represented by an effect closure can be analyzed through a $\lambda_{\mathtt{fe}}$ effect pattern matching expression, enabling effect-based dispatch to naturally support effect-guided programming.

[Effect Realization] The dual of effect reflection is effect realization: the **realize** $\mathtt{x}$ expression evaluates the passenger expression of the effect closure $\mathtt{x}$.

Foundationally, effect reflection and effect realization are the introduction and elimination of first-class effects, in the form of effect closures. As effect closures may cross modularity boundaries, it can be viewed as "effect-carrying code."

The direct benefit of $\lambda_{\mathtt{fe}}$ is its support in flexible effect-guided programming. For example, thread scheduling in concurrent programs is a prolific area of research [29, 45, 46]. With $\lambda_{\mathtt{fe}}$, programmers can flexibly develop a variety of thread management strategies. We will further demonstrate a broad range of applications beyond thread scheduling in §8 and §B. Overall, a variety of meta-level designs currently "hidden" behind the compiler and language runtime are now in the hands of programmers.

With great power comes great responsibility[1]. The grand challenge of designing a flexible programming model lies in

[1] Marvel's Spiderman

```
  ------------------- Server -------------------
1 let buf = ref_r 0 in
2  let scheduler = λ x1:exact, x2:exact.
3      let (p, c) = effcase x1:
4                     | EC(l ∼ u) where wr_r <<: l => (x1, x2)
5                     | default                   => (x2, x1) in
6       {wr_r <<: c |−> wr_r <<: p} realize p; realize c
  ------------------- Client -------------------
7  let reader = (query !buf) in
8   let writer = (query buf := 1) in
9    scheduler reader writer
```

notation	meaning
query e	effect reflection
effcase x : $\overline{\texttt{T}\ \textbf{where}\ P \Rightarrow \texttt{e}}$	predicated effect dispatch
realize e	effect realization
x ∼ y	lower bound effect x & upper bound y
EC(x ∼ y)	effect closure type
x <<:y	effect x is a subset of y
|−>	logical implication
$\{P\}$e	refinement type
$\mathbf{wr}_r$	write effect to region r
exact	lower and upper bounds are equal

Figure 1. A Producer-First Scheduler. (The New Notations Introduced by First-Class Effects Are Explained on the Right.)

principled and precise reasoning. The core technical development of $\lambda_{\texttt{fe}}$ is a type system with a number of features. First, effect reflection is designed through dynamic typing, resulting in a hybrid type-and-effect system [35] that employs run-time information to improve precision. Second, $\lambda_{\texttt{fe}}$ provides static guarantees to application-specific effect management properties through refinement types, promoting "correct-by-design" effect-guided programming. Third, $\lambda_{\texttt{fe}}$ computes not only the over-approximation of effects, the *may*-effect, but also their under-approximation, the *must*-effect. The duality unifies the common theme of permission [24] *vs.* obligation [7] in effect reasoning. Fourth, we establish a stronger notion of soundness called *trace consistency*, defined in terms of how the effect and the *trace* (the "post-execution effect") correspond. To maintain trace consistency, we introduce a notion of *polarity* to predicates defined over effects, providing a general solution to a long-standing problem in effect systems: reasoning about *non-monotone* effect operators [6, 38].

In summary, this paper makes the following contributions:

- It describes novel programming abstractions to support computational effects as first-class values. Expressive features such as effect reflection, predicated effect inspection, and effect realization, are designed to maintain the lifecycle of $\lambda_{\texttt{fe}}$ in the form of effect closures.
- It develops a sound hybrid type-and-effect system where dynamic typing is enabled as part of effect reflection, and static refinement typing is enabled by predicated effect analysis. The type system is further endowed with double-bounded effects, where may-analysis and must-analysis are performed in effect reasoning.
- It introduces a stronger notion of soundness property, trace consistency, to enable discipline first-class effects programming. Thanks to a polarity-based effect reasoning, $\lambda_{\texttt{fe}}$ enjoys trace consistency even in challenging scenarios such as supporting non-monotone effect operators.
- It demonstrates the potentially broad range of applications of first-class effects in effect-guided programming, such as effect-aware scheduling, version consistent dynamic software update, data zeroing, cooperative multi-threading, program testing, and algorithmic speculation.

2. Motivation and Design Decisions

To motivate, we illustrate how $\lambda_{\texttt{fe}}$ may help programmers implement a simple ordering strategy for thread scheduling, write-before-read, shown in Figure 1. The `scheduler` aims at executing the "producer" task (*i.e.*, the one that writes to the region `r`) first, given the two input tasks `x1` and `x2`. The two tasks are `!buf` and `buf := 1` respectively, created by the Client side of the program (lines 7-9). According to a well-known scheduling strategy for multi-threaded programs [34], a data producer should be scheduled before its consumer. For this program, the programmer may wish `writer` to be scheduled before the `reader`, regardless of whether `scheduler reader writer` appears on line 9, or `scheduler writer reader`.

In $\lambda_{\texttt{fe}}$, the **query** expression can retrieve the effects of the tasks, which were defined on lines 7-8. The resulting effects are analyzed via the **effcase** pattern matching construct. For example, on line 4, the case is matched if the *must*-effect of `x1` writes to region `r`, noted $\mathbf{wr}_r$ <<: `l`. Effects in $\lambda_{\texttt{fe}}$ have both *lower* and *upper* bounds, *e.g.*, `x1` has must-effect `l` and may-effect `u`, whose type is noted as EC(`l` ∼ `u`). The effect is realized on line 6, leading to the evaluation of the passenger expression. Finally, to guarantee that the write-before-read ordering is correctly implemented, the `scheduler` uses the refinement type {$\mathbf{wr}_r$ <<:`c` |−>$\mathbf{wr}_r$ <<:`p`}, which reads that if `c` writes to `r`, then `p` must also write to `r`.

2.1 Challenges

This simple program highlights a number of programming challenges in effect-guided programming:

- [The Need for Dynamic Effects] The `scheduler` and its client could be deployed across modularity boundaries, such as on different machines or OS domains. Even if it is easy to precisely specify the effects of the two tasks `!buf` and `buf := 1` on the client side, any practical `scheduler` should make no assumption on what the effects of `x1` and `x2` are. The more general case is that the `scheduler` takes a set of tasks as arguments.
- [Effect-Carrying Code Support] For programs where the effectful expression and its computational effect coexist in one program, principled design in both

programming abstraction and typing is required. For example, the runtime representation of first-class effects — such as the `reader` and `writer` in the example — matters. The relationship between the expressions manipulating first-class effects — such as **query** and **realize** — also requires careful type language design.

- [CUSTOM CORRECTNESS-BY-DESIGN] Different applications may have different safety criteria. In addition to the more mundane goal of providing precise effects, a feature highly desirable in effect-guided programming is to provide static guarantees to custom effect management. In the example, the programmer wishes to ensure that the program indeed has implemented the write-before-read strategy.
- [TRACE-EFFECT CORRESPONDENCE] Dynamic effect querying is tantamount to dynamic typing in a type-and-effect system. Enforcing soundness is a non-trivial task when dynamic typing is mixed with static typing [35]. For instance, an effect-guided program may be written in a way that says if the effect of the task `x1` is a superset of the effect **wr**$_r$, then execute `x1` first. Intuitively, what the programmer indeed means is that the *trace* — informally, the "post-evaluation effect" — of `x1` is a superset of **wr**$_r$. Unfortunately, effects are pre-evaluation and traces are post-evaluation, and the two do not always correspond. A well-designed system should disallow the surprising behavior where `x1` is executed first when the traces do not conform to supersetting but the effects do.

2.2 The Need for Dynamic Effect Support

We address the first challenge through two programming abstractions: the **query** and the **effcase** expressions. The **query** `e` expression plays an interesting role in effect reasoning: it enables dynamic effect reasoning, *i.e.*, a lightweight type derivation at run time. We further propose an optimized operational semantics where full-fledged dynamic typing is unnecessary. This is in contrast to dynamic effect systems [2, 4, 53] which compute the effects of `e` by collecting the traces when evaluating `e`. In this light, $\lambda_{\mathtt{fe}}$ is not an *a posteriori* effect monitoring system, and the reflected effects are still a sound and conservative approximation of the trace.

Dynamic typing allows runtime information to be used in effect computation, and hence improves the precision of effect reasoning. For example, for the code below, with dynamic typing, $\lambda_{\mathtt{fe}}$ is capable of computing both the must-effect and may-effect of the task `w` as writing to region `r`. Instead, the more conservative may-effect "writing both to `r` and `r0`" and must-effect $\emptyset$, are likely to be computed by a purely static effect system.

```
1 let client = λ buf.
2      let w = (query buf := 1) in
3        effcase w:
4        | EC(l1 ∼ u1) where wrr <<: l1 => realize w
5  client (if 1 > 0 then refr 0 else refr0 0)
```

The predicate associated with the case on line 4 says that the task must write to `r`. Thanks to the dynamic typing, the case is matched, but a static system, if equipped with the **effcase** expression, would not match the case.

2.3 Effect-Carrying Code Support

With the dual **query/effcase** design, effect querying is decoupled from effect analysis and effect-based decision making. This is useful in practice, because querying (dynamic typing) may incur runtime overhead, and the decoupling allows programmers to decide when the query should happen. For example, on line 7 and 8, the programmer says that the client should shoulder the overhead of effect query, not the server. In a similar vein, such a design allows effect of an expression to be queried once and used multiple times.

We represent the first-class effect value as an *effect closure*, a combination of a passenger expression and its effect. An effect closure can be passed across modular boundary, *e.g.*, line 9. Ultimately, we use the **realize** `x` expression to evaluate the passenger expression of the effect closure `x`, *e.g.*, line 6. Another obvious choice is to represent the effect value just as a type. The opportunity such a design misses out on is a common idiom in effect-guided programming: the reason why programmers wish to query and analyze an effect in the first place is to *evaluate the expression the aforementioned effect abstractly represents*. For example, the reason we perform effect analysis over the passenger expression `e` is to run `e` at the opportune moment.

In other words, the **query** expression in first-class effects can also be viewed as a simple form of reflection whereas the **realize** is analogous to reification. To the best of our knowledge, this is a novel design in effect reasoning systems.

2.4 Custom Correctness-By-Design

To address the third challenge, we provide static guarantees for custom-defined application-specific predicates over first-class effects through a decidable refinement type system. Our refinement type system design is intimately linked to our programming abstraction of effect analysis: the case analysis of the **effcase** expression is *predicated* [20, 39]. For example, we are capable of typechecking the program with the refinement type on line 6, thanks to the predicates associated with the **effcase** cases from lines 4-5.

2.5 Trace-Effect Correspondence

To meet the fourth challenge, we define the notion of *trace consistency*. The crucial question in $\lambda_{\mathtt{fe}}$ is whether the static guarantees represented in the form of refinement types matter for the program run-time behavior, and if so, *what they say about the run-time behavior*. In $\lambda_{\mathtt{fe}}$, for any well-typed expression whose refinement type has a predicate defined over effects, the "corresponding" predicate — identical except that every occurrence of the effect is replaced with corresponding memory accesses when said effect is realized — still holds. For example, if the refinement type {**wr**$_r$ <<: `c`

$|\!-\!>$ **wr**$_r$ <<: p}, type checks, and the memory traces of evaluating c and p are f_c and f_p respectively, then if f_c contains a write to r, then f_p must contain a write to r.

We maintain the trace consistency through two interesting features: double-bounded effects and polarity reasoning. To illustrate, consider a simple example:

EXAMPLE 2.1. *The following program fails to typecheck in $\lambda_{\mathtt{fe}}$, and rightfully so because it does not fulfill the write-before-read ordering. The main reason is that the must-effect should be used in the case analysis instead of the may-effect on line 4,* i.e., ***where wr**$_r$ <<: l. The may-effects of the* reader *is **wr**$_r$, thus the case on line 4 is matched and the* reader *is evaluated first.*

```
1 let buf = ref_r 0 in
2  let scheduler = λ x1, x2.
3    let (p, c) = effcase x1:
4                |EC(l ~ u) where wr_r <<: u => (x1, x2)
5                |default                  => (x2, x1) in
6     {wr_r <<: c |-> wr_r <<: p} realize p; realize c in
7   let reader = (query if 0 < 1 then buf := 1) in
8    let writer = (query buf := 1) in
9     scheduler reader writer
```

To prevent the misuse of the must- or may-effect, such as the one in the example above, our type system labels each n-arity custom predicate with n polarities, one for each argument. To illustrate, consider the operator <<:, $\lambda_{\mathtt{fe}}$ assigns the RHS of <<: a $+$ polarity to indicate that a must-effect should be used and the RHS a $-$. Examples for other polarities, such as $-$ and $\mathfrak{i}$, are shown in the table below.

polarity	name	example	source
$-$	monotone decreasing	LHS of <<:	may-effect
$+$	monotone increasing	RHS of <<:	must-effect
$\mathfrak{i}$	invariant	==	may equals must

Figure 2. Polarities for Predicates.

By carefully regulating the interaction between may-must effects and predicate polarity, our type system is capable of maintaining trace consistency. The program above fails to type check because the must-effect should be used *i.e.*, **where wr**$_r$ <<:l. Intuitively, the must-effect l is a subset of the trace $f_{\mathtt{x1}}$ and $f_{\mathtt{x1}}$ is a subset of the may-effect u. Therefore, if this case is matched, **wr**$_r$ is a subset of $f_{\mathtt{x1}}$, *i.e.*, x1 will write to r.

Additional Examples Scheduling is one of many applications where effect-guided programming may make a positive impact on. Additional examples will be found in §8 and §B.

3. $\lambda_{\mathtt{fe}}$: a Calculus with First-Class Effects

The abstract syntax of $\lambda_{\mathtt{fe}}$ with first-class effects, but without refinement types, is defined in Figure 3. We defer the discussion of refinement type with effect polarities to §5. Our calculus is built on top of an imperative region-based λ calculus. Expressions are mostly standard, except the constructs for effect management. As parallel programs serve as an important application domain of first-class effects, we support the parallel composition expression e||e. We model branching and boolean values explicitly, because they are useful to highlight features such as double bounded effects. The sequential composition e; e$'$ in the examples is the sugar form of **let** x $=$ e **in** e$'$.

$$\begin{array}{rcll}
\mathtt{e} & ::= & \mathtt{b} \mid \lambda\mathtt{x}:\mathtt{T}.\mathtt{e} \mid \mathtt{x} \mid \mathtt{e}\ \mathtt{e} & \textit{expressions}\\
 & \mid & \mathbf{let}\ \mathtt{x}=\mathtt{e}\ \mathbf{in}\ \mathtt{e} \mid \mathtt{e}||\mathtt{e} & \\
 & \mid & \mathbf{ref}\ \rho\ \mathtt{T}\ \mathtt{e} \mid\ !\mathtt{e} \mid \mathtt{e}:=\mathtt{e} & \textit{reference}\\
 & \mid & \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{e}\ \mathbf{else}\ \mathtt{e} & \textit{branching}\\
 & \mid & \mathbf{effcase}\ \mathtt{x}=\mathtt{e}:\overline{\mathtt{T}\ \mathbf{where}\ P\Rightarrow\mathtt{e}} & \textit{effect dispatch}\\
 & \mid & \mathbf{query}\ \mathtt{e} & \textit{effect reflection}\\
 & \mid & \mathbf{realize}\ \mathtt{e} & \textit{effect realization}\\
P & ::= & \mathtt{b} \mid P\wedge P \mid P\vee P \mid \neg P \mid \mathbb{P}\,\overline{\sigma} & \textit{predicate}\\
g & ::= & \alpha & \textit{type variable}\\
 & \mid & \gamma & \textit{region variable}\\
 & \mid & \varsigma & \textit{effect variable}\\
\mathtt{T} & ::= & \mathbf{Bool} \mid \alpha \mid \mathbf{Ref}_\rho\ \mathtt{T} & \textit{type}\\
 & \mid & \mathtt{T} \xrightarrow{\sigma\sim\sigma'} \mathtt{T}' & \textit{function type}\\
 & \mid & \mathbf{EC}(\mathtt{T},\sigma\sim\sigma') & \textit{effect type}\\
\rho & ::= & \mathtt{r} \mid \gamma \mid \overline{\rho} & \textit{region}\\
\sigma & ::= & \pi_\rho \mid \varsigma \mid \overline{\sigma} & \textit{effect}\\
\pi & ::= & \mathbf{init} \mid \mathbf{rd} \mid \mathbf{wr} & \textit{allocation, read and write}\\
\mathtt{b} & ::= & \textit{true} \mid \textit{false} & \textit{boolean}
\end{array}$$

Figure 3. $\lambda_{\mathtt{fe}}$ Abstract Syntax. (In this Paper, Notation $\overline{\bullet}$ Represents a Set of $\bullet$ Elements).

Effect management. consists of the key abstractions described in §2. Expression **query** e dynamically computes the effect of expression e. The result of a query is an effect closure. Programmers can inspect a closure with **effcase** x $=$ e : $\overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{e}}$. The expression evaluates e to an effect closure, upon which predicated pattern matching is performed. The expression also introduces a variable x, and binds it to e. Such a variable can be used in the **realize** expression to refer to which effect closure is to be realized, and may appear in refinement types to specify custom static guarantees over effects. If the expression e is a variable, *e.g.*, **effcase** x $=$ x$_0$: $\overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{e}}$, we shorten it to **effcase** x$_0$: $\overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{e}}$, as is the case in the examples. This effect analysis expression pattern-matches the closure against the type patterns $\overline{\mathtt{T}}$. P is a type constraint to further refine pattern matching. It supports connectives of proposition logic, together with the atomic n-ary form $\mathbb{P}\ \overline{\sigma}$, left abstract, which can be concretized into different forms for different concrete languages. Please find examples of different instantiations of $\mathbb{P}$ in §8.

Effects and Effect Types. Effects are region accesses and have the form π_ρ, representing an access right π to values in region ρ. Access rights include allocation **init**, read **rd** and write **wr**.

Compared with existing effect systems, our system supports both must- and may-effects. Function types $\mathtt{T} \xrightarrow{\sigma\sim\sigma'} \mathtt{T}'$ specifies a function from $\mathtt{T}$ to $\mathtt{T}'$ with must-effect σ and may-effect σ' as the effects of the function body.

Effect closure type has the form $\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')$. The value it represents produces must-effect σ, may-effect σ' upon realization, and the realized expression has type $\mathtt{T}$. When $\mathtt{T}$ is not used (*e.g.*, in Figure 1), we shorten it as $\mathbf{EC}(\sigma \sim \sigma')$.

Regions. The domain of regions is the disjoint union of a set of constants $\mathtt{r}$. The region abstracts memory locations in which it will be allocated at runtime. Our notion of region is standard [37, 52]. In $\lambda_{\mathtt{fe}}$, allocation sites are explicitly labelled with regions. Region inference is feasible [21, 24], an issue orthogonal to our interest.

In $\lambda_{\mathtt{fe}}$, type α, effect ς, and region γ variables are cumulatively referred to as "pattern variables", and we use a metavariable g for them. A type variable α can be used in the **effcase** expressions to match any type $\mathtt{T}$, given that the constraint P is satisfied if α is substituted with $\mathtt{T}$, similar for effect and region variables.

Before we proceed, let us provide some notations and convenience functions used for the rest of the paper. Functions *dom* and *rng* are the conventional domain and range functions. Substitution θ maps type variables α to types $\mathtt{T}$, region variables γ to regions ρ, and effect variables ς to effects σ. Comma is used for sequence concatenation.

4. A Base Type System with Double-Bounded Effects

The key innovations of our type system design are twofold. First, it uses double bounded types to capture must-may effects. Second, it employs refinement types to fulfill hybrid effect reasoning. We present double-bounded effects in this section, and delay refinement types to §5.

4.1 Subtyping

Relation $\mathtt{T} <: \mathtt{T}'$ says $\mathtt{T}$ is a subtype of $\mathtt{T}'$ defined in Figure 4. The subtyping relation is reflexive and transitive. Reference **ref** types follow invariant subtyping, except that the regions in the **ref** types follow covariant subtyping.

The highlight of the subtyping relation lies in the treatment of the must-may effects. In (sub-FUN), observe that must-effects and may-effects follow opposite directions of subtyping: may-effects are covariant whereas must-effects are contravariants. Intuitively, for a program point that expects a function that *must* produce effect σ, it is always OK to be provided with a function that *must* produce a "superset effect" of σ. On the flip side, for a program point that expects a function that *may* produce effect σ, it is always OK to be provided with a function that *may* produce a "subset effect" of σ. As expected, effect subsumption in our system — the "superset effect" and the "subset effect" — is supported through set containment over σ elements. Effect closure types follow a similar design, as seen in (sub-EC).

$\boxed{\text{Subtyping: } \mathtt{T} <: \mathtt{T}'}$

$$(sub\text{-REFL})\quad \mathtt{T} <: \mathtt{T} \qquad (sub\text{-TRANS})\quad \frac{\mathtt{T} <: \mathtt{T}'' \qquad \mathtt{T}'' <: \mathtt{T}'}{\mathtt{T} <: \mathtt{T}'} \qquad (sub\text{-REF})\quad \frac{\rho \subseteq \rho'}{\mathbf{Ref}_{\rho}\ \mathtt{T} <: \mathbf{Ref}_{\rho'}\ \mathtt{T}}$$

$$(sub\text{-FUN})\quad \frac{\mathtt{T}'_x <: \mathtt{T}_x \qquad \mathtt{T} <: \mathtt{T}' \qquad \sigma_2 \subseteq \sigma_0 \qquad \sigma_1 \subseteq \sigma_3}{\mathtt{T}_x \xrightarrow{\sigma_0\sim\sigma_1} \mathtt{T} <: \mathtt{T}'_x \xrightarrow{\sigma_2\sim\sigma_3} \mathtt{T}'}$$

$$(sub\text{-EC})\quad \frac{\mathtt{T} <: \mathtt{T}' \qquad \sigma_2 \subseteq \sigma_0 \qquad \sigma_1 \subseteq \sigma_3}{\mathbf{EC}(\mathtt{T}, \sigma_0 \sim \sigma_1) <: \mathbf{EC}(\mathtt{T}', \sigma_2 \sim \sigma_3)}$$

Figure 4. The Subtyping Relation.

Our covariant design of may-effects and contravariant design of must-effects on the high level is aligned with the intuition that along the data flow path, the dual bounds of a function, or those of a first-class effect closure may potentially be "loosened."

4.2 Type Checking

Type environment Γ maps variables to types:

$$\Gamma \quad ::= \quad \overline{\mathtt{x} \mapsto \mathtt{T}}$$

Notation $\Gamma(\mathtt{x})$ denotes $\mathtt{T}$ if the rightmost occurrence of $\mathtt{x} : \mathtt{T}'$ for any $\mathtt{T}'$ in Γ is $\mathtt{x} : \mathtt{T}$.

Type checking is defined through judgment $\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$, defined in Figure 5. The judgment says under type environment Γ, expression $\mathtt{e}$ has type $\mathtt{T}$, must-effect σ and may-effect σ'. Subtyping is represented in the type checking process through (T-SUB), which follows the same pattern to treat must-may effects as in function subtyping and effect closure subtyping.

Effect Bound Reasoning. If effect bounds are "loosened" along the data flow path as we discussed, the interesting question is when the bounds are "tightened". To answer this question, observe that traditional effect systems can indeed be viewed as a (degenerate) double-bounded effect system, where the must-effect is always the empty set.

Our type system on the other hand computes the must-effect along the type checking process. Note that in (T-IF), the must-effect of the branching expression is the *intersection* of the must-effects of the **then** branch and the **else** branch, unioned with the must-effect of the conditional expression. For example, given an expression as follows and that `val` is in region $\mathtt{r}$, the must- and may-effects are $\{\mathbf{rd}_r\}$ and $\{\mathbf{rd}_r, \mathbf{wr}_r\}$ respectively:

```
if x > 0 then !val else val := !val + 1
```

The must-effect of the **effcase** expression is computed in an analogous fashion, as shown in the (T-EFFCASE) rule.

Typing Effect Operators. (T-QUERY) shows the expression to introduce an effect closure — the **query** expression — is typed as an effect closure type, including both the

$\boxed{\text{Type Checking: } \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}$

$$\text{(T-EFFCASE)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset \qquad \exists \theta\ .(\theta \mathbf{EC}(\mathtt{T}_i, \sigma_i \sim \sigma'_i) <: \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')), \text{for all } i \in \{1 \dots n\} \\ \exists \varsigma\ .\ \mathtt{T}_n = \varsigma \wedge P_n = \mathit{true} \qquad \Gamma, \mathtt{x} \mapsto \mathbf{EC}(\mathtt{T}_i, \sigma_i \sim \sigma'_i) \vdash \mathtt{e}_i : \mathtt{T}, \sigma''_{\mathtt{i}} \sim \sigma'''_{\mathtt{i}}, \text{for all } i \in \{1 \dots n\}\end{array}}{\Gamma \vdash \textbf{effcase}\ \mathtt{x} = \mathtt{e} : \overline{\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')\ \textbf{where}\ P \Rightarrow}\ \mathtt{e} : \mathtt{T}, \cap_{i \in \{1 \dots n\}} \sigma''_i \sim \cup_{i \in \{1 \dots n\}} \sigma'''_i}$$

$$\text{(T-QUERY)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}{\Gamma \vdash \textbf{query}\ \mathtt{e} : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset} \qquad \text{(T-REALIZE)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathbf{EC}(\mathtt{T}, \sigma_0 \sim \sigma_1), \sigma_2 \sim \sigma_3}{\Gamma \vdash \textbf{realize}\ \mathtt{e} : \mathtt{T}, \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3}$$

$$\text{(T-ABS)}\quad \frac{\Gamma, \mathtt{x} \mapsto \mathtt{T} \vdash \mathtt{e} : \mathtt{T}', \sigma \sim \sigma'}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} : \mathtt{T} \xrightarrow{\sigma \sim \sigma'} \mathtt{T}', \emptyset \sim \emptyset} \qquad \text{(T-APP)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \xrightarrow{\sigma_0 \sim \sigma_1} \mathtt{T}', \sigma_2 \sim \sigma_3 \quad \Gamma \vdash \mathtt{e}' : \mathtt{T}, \sigma_4 \sim \sigma_5}{\Gamma \vdash \mathtt{e}\ \mathtt{e}' : \mathtt{T}', \sigma_0 \cup \sigma_2 \cup \sigma_4 \sim \sigma_1 \cup \sigma_3 \cup \sigma_5} \qquad \text{(T-BOOL)}\quad \Gamma \vdash \mathtt{b} : \mathbf{Bool}, \emptyset \sim \emptyset$$

$$\text{(T-SUB)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \qquad \mathtt{T} <: \mathtt{T}' \\ \sigma_0 \subseteq \sigma \qquad \sigma' \subseteq \sigma_1\end{array}}{\Gamma \vdash \mathtt{e} : \mathtt{T}', \sigma_0 \sim \sigma_1} \qquad \text{(T-LET)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma_0 \sim \sigma_1 \qquad \Gamma, \mathtt{x} \mapsto \mathtt{T} \vdash \mathtt{e}' : \mathtt{T}', \sigma_2 \sim \sigma_3}{\Gamma \vdash \textbf{let}\ \mathtt{x} = \mathtt{e}\ \textbf{in}\ \mathtt{e}' : \mathtt{T}', \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3} \qquad \text{(T-GET)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathbf{Ref}_\rho\ \mathtt{T}, \sigma \sim \sigma'}{\Gamma \vdash !\,\mathtt{e} : \mathtt{T}, \sigma \cup \mathbf{rd}_\rho \sim \sigma' \cup \mathbf{rd}_\rho}$$

$$\text{(T-REF)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}{\Gamma \vdash \textbf{ref}\ \rho\ \mathtt{T}\ \mathtt{e} : \mathbf{Ref}_\rho\ \mathtt{T}, \sigma \cup \mathbf{init}_\rho \sim \sigma' \cup \mathbf{init}_\rho} \qquad \text{(T-SET)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathbf{Ref}_\rho\ \mathtt{T}, \sigma_0 \sim \sigma_1 \qquad \Gamma \vdash \mathtt{e}' : \mathtt{T}, \sigma_2 \sim \sigma_3}{\Gamma \vdash \mathtt{e} := \mathtt{e}' : \mathtt{T}, \sigma_0 \cup \sigma_2 \cup \mathbf{wr}_\rho \sim \sigma_1 \cup \sigma_3 \cup \mathbf{wr}_\rho} \qquad \text{(T-VAR)}\quad \frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}, \emptyset \sim \emptyset}$$

$$\text{(T-PARA)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma_0 \sim \sigma_1 \qquad \Gamma \vdash \mathtt{e} : \mathtt{T}', \sigma_2 \sim \sigma_3}{\Gamma \vdash \mathtt{e} || \mathtt{e}' : \mathtt{T}', \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3} \qquad \text{(T-IF)}\quad \frac{\Gamma \vdash \mathtt{e} : \mathbf{Bool}, \sigma_0 \sim \sigma_1 \qquad \Gamma \vdash \mathtt{e}_0 : \mathtt{T}, \sigma_2 \sim \sigma_3 \qquad \Gamma \vdash \mathtt{e}_1 : \mathtt{T}, \sigma_4 \sim \sigma_5}{\Gamma \vdash \textbf{if}\ \mathtt{e}\ \textbf{then}\ \mathtt{e}_0\ \textbf{else}\ \mathtt{e}_1 : \mathtt{T}, \sigma_0 \cup (\sigma_2 \cap \sigma_4) \sim \sigma_1 \cup \sigma_3 \cup \sigma_5}$$

Figure 5. Typing Rules.

static type of the to-be-dynamically-typed expression, and its double-bounded effects as reasoned by the static system. In other words, even though first-class effects will be computed at runtime based on information garnered from (the more precise) dynamic typing, our static type system still makes its best effort to type this first-class value, instead of viewing it as an opaque "top" type of the effect closure kind.

The dual of the **query** expression is the **realize** expression. Intuitively, this expression "eliminates" the effect closure, and evaluates the passenger expression. (T-REALIZE) is defined to be consistent with this view. It says that the expression should have the type of the passenger expression, and the effects should include both those of the expression that will evaluate to the effect closure, and those of the passenger expression.

The (T-EFFCASE) rule shows that the **effcase** expression predictably follows the pattern matching semantics. The expression to be analyzed must represent an effect closure. To avoid unreachable patterns, the type system ensures every type pattern is indeed satisfiable through substitution and subtyping. In addition, $\lambda_{\mathtt{fe}}$ requires that the last pattern be a pattern variable, which matches any type, serving as the explicit "**`default`**" clause [1].

Standard Expressions. Other typing rules are mostly conventional. Store operations (T-REF), (T-GET) and (T-SET) compute initialization **init**, read **rd** and write **wr** effects, respectively. The typing of parallel composition is standard.

5. The Full-Fledged System

The type system in Figure 5 does not provide any static guarantees for expressions guarded by the predicate P in the **effcase** expression. For example, in resource-aware scheduling (Figure 1), the programmer may wish to be provided with the *static* guarantee that the order of buffer access is preserved. We support this refined notion of reasoning through refinement types.

We extend the grammar of our language, in Figure 6, to allow the programmers to associate an expression with a refinement type, denoting that the corresponding expression must satisfy the predicate in the refinement type through static type checking.

A refinement type $\mathfrak{T}$ takes the form of $\{\mathtt{T}, \sigma \sim \sigma' | P\}$, where predicate P is used to refine the base type $\mathtt{T}$ and effects $\sigma \sim \sigma'$, a common notation in refinement type systems [12, 25, 44]. When $\mathtt{T}$, σ and σ' are not referred to in other parts in the refinement type, we shorten the refinement as $\{P\}$, *e.g.*, in the write-before-read example in Figure 1, $\{\mathbf{wr}_r <<: \mathtt{c} \mid\!-\!> \mathbf{wr}_r <<: \mathtt{p}\}$. We extend the subtyping relation with one additional rule, (*subr*-REFINE). Here, subtyping of refinement types is defined as the logical implication $|\!-\!>$ of the predicates of the two types.

$$
\begin{array}{lll r}
\mathtt{e} & ::= & \ldots \mid \mathfrak{T}\,\mathtt{e} & \textit{extended expression} \\
\mathfrak{T} & ::= & \{\mathtt{T}, \sigma \sim \sigma' | P\} & \textit{refinement type} \\
\mathtt{T} & ::= & \ldots \mid \mathfrak{T} & \textit{type} \\
\Delta & ::= & \overline{\varsigma \mapsto \mathscr{V}} & \textit{polarity environment} \\
\mathscr{V} & ::= & + \mid - \mid * \mid \mathfrak{i} & \textit{polarity}
\end{array}
$$

$\boxed{\text{Subtyping: } \tau <: \tau'}$

$$
(subr\text{-REFINE}) \quad \frac{P \mid\!-\!> P'}{\Gamma \vdash \{\mathtt{T}, \sigma \sim \sigma' | P\} <: \{\mathtt{T}, \sigma \sim \sigma' | P'\}}
$$

For all other ($subr$-*) rules, each is isomorphic to its counterpart (sub-*) rule in Figure 4.

Figure 6. $\lambda_{\mathtt{fe}}$ Extension with Refinement Types.

5.1 Polarity Support

Polarity environment Δ, which will be used in type checking, maps effect variables ς to polarities $\mathscr{V}$. $\mathscr{V}$ can either be contravariant $+$, covariant $-$, invariant $\mathfrak{i}$ and bivariant $*$. Intuitively, the $+$ comes from the must-effect, $-$ comes from the may-effect. If must- and may-effects are exactly the same, it induces invariant $\mathfrak{i}$, *e.g.*, the predicate ==, which requires the effects of its LHS and RHS to be equal in Figure 21. If ς appears in both must- and may-effects, but the effects are not the same, ς will be bivariant. The subsumption relations of the variances form a lattice, defined in Figure 7, with the join $\sqcup$ going "up". Intuitively, an $\mathfrak{i}$ can appear in a position where $+$ is required, thus $\mathfrak{i}$ is a "subtype" of $+$.

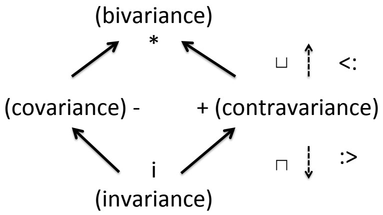

Figure 7. Polarity Lattice.

For a predicate of arity n, we say its position j ($1 \leq j \leq n$) to have contravariant polarity $+$ when effect subsumption of argument j is aligned with predicate implication, *i.e.*, an application of this predicate with argument j being σ always implies a predicate application identical with the former except argument j being σ' and $\sigma \subseteq \sigma'$, shown in (MONO-$\boxed{\uparrow}$) in Figure 9, where the $\mathbb{V}(\mathbb{P}, j)$ (Figure 8) notation gets the j^{th} polarity of the predicate $\mathbb{P}$, *e.g.*, $\mathbb{V}(<<:, 0)$ will return the polarity of LHS of <<:, *i.e.*, $-$ and $\mathbb{V}(<<:, 1)$ will return RHS, *i.e.*, $+$. Similarly, we say the position j of a predicate to have covariant polarity $-$ if an application of this predicate with argument j being σ always implies a predicate application identical with the former except argument j being σ' and $\sigma' \subseteq \sigma$, as (MONO-$\boxed{\downarrow}$). For non-monotone predicates (*e.g.*, ==), the polarities $+$ and $-$ fall short:

EXAMPLE 5.1. (Effect Invariant for Non-monotone Effect Predicate) *Programmers wish to check that the effects of two expressions are equal, line 4. The non-monotone (for both LHS and RHS) predicate == is satisfied for the call on line 5, but challenging for a system with co- or contravariant polarities alone.*

```
1 let buf = ref_r -1 in
2  let fun = λ x:exact, y:exact. effcase x, y:
3               | EC(l_0 ~ l_0),EC(l_1 ~ l_1)
4                   where l_0 == l_1 => {y == x} x; y in
5   fun (query !buf) (query !buf);
6   fun (query buf := 0) (query !buf)
```

$\mathbb{P}$	LHS ($j = 0$)	RHS ($j = 1$)
<<:	$-$(may)	$+$(must)
/<<:	$+$(must)	$-$(may)
#	$-$(may)	$-$(may)
==	$\mathfrak{i}$(may and must)	$\mathfrak{i}$(may and must)

Figure 8. Polarities for Client Predicates ($\mathbb{V}(\mathbb{P}, j)$).

$\boxed{\text{Predicate Implication: } P \mid\!-\!> P}$

$$
(\text{MONO-}\boxed{\uparrow}) \quad \frac{\mathbb{V}(\mathbb{P}, j) = + \qquad \sigma_j \subseteq \sigma'_j}{\mathbb{P}\,\overline{\sigma}\sigma_j\overline{\sigma'} \mid\!-\!> \mathbb{P}\,\overline{\sigma}\sigma'_j\overline{\sigma'}}
\qquad
(\text{MONO-}\boxed{\downarrow}) \quad \frac{\mathbb{V}(\mathbb{P}, j) = - \qquad \sigma'_j \subseteq \sigma_j}{\mathbb{P}\,\overline{\sigma}\sigma_j\overline{\sigma'} \mid\!-\!> \mathbb{P}\,\overline{\sigma}\sigma'_j\overline{\sigma'}}
$$

$$
(\text{IMP-}\wedge 0) \quad P \wedge P' \mid\!-\!> P
\qquad
(\text{IMP-}\vee 0) \quad P \mid\!-\!> P \vee P'
\qquad
(\text{IMP-REFL}) \quad P \mid\!-\!> P
$$

$$
(\text{IMP-}\wedge 1) \quad P \wedge P' \mid\!-\!> P'
\qquad
(\text{IMP-}\vee 0) \quad P' \mid\!-\!> P \vee P'
\qquad
(\text{IMP-TRANS}) \quad \frac{P \mid\!-\!> P' \qquad P' \mid\!-\!> P''}{P \mid\!-\!> P''}
$$

Figure 9. Predicate Implication $\mid\!-\!>$.

Note that the equivalent of the may-effects (or must-effect) of both sides does not guarantee the equivalent of the traces (runtime memory accesses), *e.g.*, for the following code, the two parameters (line 8) are the same, but their runtime traces are not the same.

```
7 let same = (query if !buf < 0 then buf := 0) in
8  fun same same
```

The **exact** annotation solves this problem. This annotation requires that the may- and must-effects of the expression are the same, inducing invariant. Since the trace is bounded by the same lower and upper bound effects, it is tight. Given that x==y and that both x and y have tight bounds, their traces must be equal.

The must-effect of the expression on line 7 is $\{\mathbf{rd}_r\}$, the may-effect is $\{\mathbf{rd}_r, \mathbf{wr}_r\}$, and thus its effects are not **exact** and the function call, on line 8, fails static type checking. The effects of the four queried expressions on lines 5-6 are **exact**. The calls on those two lines will type check statically. The call on line 5 will satisfy the runtime predicate == and execute the code on line 4, while the call on line 8 will not satisfy the predicate and thus not execute the code on line 4. Similarly, the **exact** annotation fulfills a similar task in Figure 1. Without **exact**, the program will not be sound.

$\boxed{\text{Refinement Type Checking: } \Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}$

$$\text{(R-REFINE)}\quad \frac{\Delta \vdash P \qquad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \qquad \llbracket\Gamma\rrbracket \mid\!-\!> P}{\Delta;\Gamma \vdash \{\mathtt{T}, \sigma \sim \sigma' | P\}\ \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}$$

$$\text{(R-EFFCASE)}\quad \frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e} : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset \qquad \exists\theta\ .(\theta\mathbf{EC}(\mathtt{T}_i, \sigma_i \sim \sigma'_i) <: \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')), \text{for all } i \in \{1 \dots n\} \\ \exists\varsigma\ .\ \mathtt{T}_n = \varsigma \wedge P_n = \emptyset \qquad \Delta_i = \Delta \sqcup \text{polar}_t(\mathtt{T}_i) \sqcup \text{polar}_e(\sigma_i \sim \sigma'_i) \\ \Delta_i \vdash P_i \qquad \Delta_i;\Gamma, \mathtt{x} \mapsto \{\mathbf{EC}(\mathtt{T}_i, \sigma_i \sim \sigma'_i), \emptyset \sim \emptyset | P_i\} \vdash \mathtt{e}_i : \mathtt{T}, \sigma''_i \sim \sigma'''_i, \text{for all } i \in \{1 \dots n\}\end{array}}{\Delta;\Gamma \vdash \textbf{effcase}\ \mathtt{x} = \mathtt{e} : \overline{\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')\ \textbf{where}\ P \Rightarrow} \mathtt{e} : \mathtt{T}, \cap_{i\in\{1\dots n\}}\sigma''_i \sim \cup_{i\in\{1\dots n\}}\sigma'''_i}$$

For all other (R-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of judgment $\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the latter rule should be substituted with $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the former.

$\boxed{\text{Type Checking Predicate: } \Delta \vdash P}$

$$\frac{\neg\Delta \vdash P}{\Delta \vdash \neg P} \qquad \frac{\Delta \vdash P \qquad \Delta \vdash P'}{\Delta \vdash P \wedge P'} \qquad \frac{\Delta \vdash P \qquad \Delta \vdash P'}{\Delta \vdash P \vee P'} \qquad \frac{\forall\sigma_j \in \overline{\sigma}\forall\varsigma \in \sigma_j \text{ s.t. } \varsigma \in dom(\Delta).\Delta(\varsigma) <: \mathbb{V}(\mathbb{P}, j)}{\Delta \vdash \mathbb{P}\,\overline{\sigma}}$$

Figure 10. Typing Rules for Checking Refinement Types.

In this presentation, our polarity modifiers are coarse-grained in that when we say an effect is **exact**, it does not allow any variance on effects to *any* region. A finer-grained approach is to allow polarity to be *region-parametric*, in that a notation such as **exact**<r> says that no variance is allowed for effects on region r, but the default (bivariance) applies for all other regions. With this finer-grained formulation, the example in Figure 1, can be updated where every occurrence of **exact** can be replaced by **exact**<r>. We here do not formalize this useful but predictable extension.

5.2 Refinement Type Checking

Refinement type checking is defined through judgment $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ shown in Figure 10, which extends the rules in Figure 5 with one additional rule (R-REFINE) for refinement typing and one adaptation rule (R-EFFCASE) for typing predicated effect analysis. The rules ensure that the pattern variables are properly used, *e.g.*, a variable ς with $-$ polarity should not appear in the position where the predicate requires $+$, such as u on line 4 in Example 2.1. (R-REFINE) requires that the predicate in the refinement type to be entailed from the predicates in the type environment. Function $\llbracket\Gamma\rrbracket$ computes the conjunction of all predicates that appear in the refinement types of Γ [13], defined in Figure 11.

$\boxed{\text{Predicate Combination: } \llbracket\Gamma\rrbracket = P}$

$$\llbracket\overline{\varsigma \mapsto \{\mathtt{T}, \sigma \sim \sigma' | P\}}\rrbracket \quad = \quad \textstyle\bigwedge_{j=1}^{n} P_j$$

$\boxed{\text{Environment Negation: } \neg\Delta = \Delta}$

$$\neg\overline{\varsigma \mapsto \mathscr{V}} \quad = \quad \overline{\varsigma \mapsto \neg\mathscr{V}}$$

$\boxed{\text{Polarity Negation: } \neg\mathscr{V} = \mathscr{V}}$

$$\begin{array}{rcl} \neg + & = & - \\ \neg - & = & + \\ \neg * & = & * \\ \neg \mathfrak{i} & = & \mathfrak{i} \end{array}$$

$\boxed{\text{Computing } \Delta \text{ from Type: polar}_t(\mathtt{T}) = \Delta}$

$$\begin{array}{r} \text{polar}_t(\mathbf{Bool}) = \emptyset \\ \text{polar}_t(\alpha) = \emptyset \\ \text{polar}_t(\mathbf{Ref}_\rho\ \mathtt{T}) = \text{polar}_t(\mathtt{T}) \\ \text{polar}_t(\mathtt{T} \xrightarrow{\sigma\sim\sigma'} \mathtt{T}') = \text{polar}_t(\mathtt{T}) \sqcup \text{polar}_t(\mathtt{T}') \sqcup \text{polar}_e(\sigma \sim \sigma') \\ \text{polar}_t(\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')) = \text{polar}_t(\mathtt{T}) \sqcup \text{polar}_e(\sigma \sim \sigma') \end{array}$$

$\boxed{\text{Computing } \Delta \text{ from Effect: polar}_e(\sigma \sim \sigma) = \Delta}$

$$\begin{array}{rcl} \text{polar}_e(\varsigma \sim \varsigma) & = & \varsigma \mapsto \mathfrak{i} \\ \text{polar}_e(\pi_\rho \sim \pi'_{\rho'}) & = & \emptyset \\ \text{polar}_e(\varsigma \cup \sigma \sim \sigma') & = & \varsigma \mapsto + \sqcup \text{polar}_e(\sigma \sim \sigma') \\ \text{polar}_e(\sigma \sim \varsigma \cup \sigma') & = & \varsigma \mapsto - \sqcup \text{polar}_e(\sigma \sim \sigma') \end{array}$$

Figure 11. Functions for Computing Effect Polarity.

The differences between (R-EFFCASE) and (T-EFFCASE) (in §4) are highlighted. We compute, via the polar function defined in Figure 11, the polarity for each new effect variable appears in the pattern matching types. An effect variable ς appearing in the must-effect, will have $+$ polarity. If ς appears in the may-effect, it will have $-$ polarity. If the must- and may-effects are the same ς, ς has $\mathfrak{i}$ polarity, otherwise ς has $*$ polarity. We use the computed polarities to check the proper use of the effect variables in each predicate, which is defined in the bottom of the same figure. For example, an effect variable ς with $+$ polarity should not appear in the position where the predicate requires $-$. Most of the checking rules are rather predictable except for the predicate negation. To check the negation, we negate the polarity environment $\neg\Delta$. For example, we require the may-effect of the

LHS and must-effect of the RHS of the predicate <<:, and for its negation /<<:, we require the must-effect of the LHS and may-effect of the RHS. The custom predicates and their polarities specifications are shown in Figure 8. The rules associate $\mathtt{x}$ with a refinement type that carries the guarded predicate, P_i in the typing environment, which will be used to check the (R-REFINE) rule.

Effect closures in $\lambda_{\mathtt{fe}}$ are immutable. This language feature significantly simplifies the design of refinement types in $\lambda_{\mathtt{fe}}$, as the interaction between refinement types and mutable features would otherwise be challenging [12].

6. Dynamic Semantics

This section describes $\lambda_{\mathtt{fe}}$'s dynamic semantics. The highlight is to support runtime effects management and highly precise effects reasoning through dynamic typing.

Semantics Objects. $\lambda_{\mathtt{fe}}$'s configuration consists of a *store* $\mathtt{s}$, an expression $\mathtt{e}$ to be evaluated, and an effects *trace* f, defined in Figure 12. These definitions are conventional. The domain of the store consists of a set of references l. Each reference cell in $\mathtt{s}$ records a value, as well as the region $\mathtt{r}$ and type $\mathtt{T}$ of the reference. The trace records the runtime accesses to regions along the evaluation, with **init**$(\mathtt{r})$, **rd**$(\mathtt{r})$, and **wr**$(\mathtt{r})$, denoting the initialization, read, and write to $\mathtt{r}$, respectively. Traces only serve a role in the soundness proofs, and thus are unnecessary in a $\lambda_{\mathtt{fe}}$ implementation. More specifically, we will show that the trace is the "realized effects" of the effects computed by $\lambda_{\mathtt{fe}}$.

The small-step semantics is defined as transition $\mathtt{s}; \mathtt{e}; f \rightarrow \mathtt{s}'; \mathtt{e}'; f'$. Given a store $\mathtt{s}$ and a trace f, the evaluation of an expression $\mathtt{e}$ results in another expression $\mathtt{e}'$, a (possibly updated) store $\mathtt{s}'$, and a trace f'. The notation $[\mathtt{x}\backslash\mathtt{v}]\mathtt{e}$ substitutes $\mathtt{x}$ with $\mathtt{v}$ in expression $\mathtt{e}$. The notation $\rightarrow^*$ represents the reflexive-transitive closure of $\rightarrow$.

We highlight the first-class effects expressions.

Effect Querying as Dynamic Typing. The (QRY) rule illustrates the essence of $\lambda_{\mathtt{fe}}$'s effect querying. An effect query produces an effect closure, which encapsulates the queried expression and its runtime type-and-effect.

Dynamic typing, defined in Figure 12, is used to compute the effects of the queried expression. The dynamic type derivation has the form $\mathtt{s}; \Delta; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$, which extends static typing with two new rules for reference value and effect closure typing.

At runtime, the free variables of the expressions will be substituted with values, *e.g.*, in (LET) and (APP). Thus, $\mathtt{e}$ in **query** $\mathtt{e}$ is no longer the same as what it was in the source program. These substituted values carry more precise types, regions, and effects information, bringing to first-class effects a highly precise notion of effects reasoning comparing to a static counterpart. Also the dynamic typing does not evaluate the queried expression to compute effects, *i.e.*, $\lambda_{\mathtt{fe}}$ is not an *a posteriori* effect monitoring system.

Applying full-fledged dynamic typing could be expensive. In §A, we provide an optimization. Observe that the difference between the static and dynamic typing (Figure 5 and Figure 12) is the types of the free variables in the environment Γ. Our insight is that we can pre-compute the dynamic effect introducing skolem type-and-effect variables for the types of the free variables, compute the effects $\sigma_0 \sim \sigma_1$ for the to-be-queried expression, and store the effects. At runtime, we substitute the skolem variables with their corresponding type and derive the effect $\sigma'_0 \sim \sigma'_1$ from $\sigma_0 \sim \sigma_1$.

Effect Realization. Dual to the effect querying rule is the realization (REAL) rule, which "eliminates" the effect closure $\langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle$ and evaluates the passenger expression $\mathtt{e}$. To show the *validity* of the effect query in first-class effects, we will prove in Theorem 7.3, that if $\mathtt{e}$ is evaluated, its effects will fall within the lower σ and upper bound σ' effects. When the evaluation terminates, it reduces to a value of type $\mathtt{T}$, specified in the closure.

Effect-Guided Programming via Predicated Effect Inspection. The (EFFC) rule analyzes the type-and-effect of the closure. It searches the first matching target types and returns the corresponding branch expression $\mathtt{e}_i$. Such type must be refined by the inner type of the effect closure, with proper "alignment" by substituting the pattern effect variables in the target type with the corresponding effects. It also requires that the substitution satisfies the target programmer-defined effect analyses predicate P_i.

Refinement expressions. The (RFMT) rule models refinement expressions. It retrieves the dynamic effects of the subexpression and checks that they are the same as specified in the refinement type and that the predicate is satisfied. The trace soundness property of our system guarantees that refinement type checking at runtime (see Theorem 7.6) always succeeds. Therefore, these runtime types checking can be treated as no-op.

Parallelism. The (PAR) rule simulates parallelism. Its treatment is standard, it nondeterministically reduces to the sequential compositions $\mathtt{e}; \mathtt{e}'$ or **let** $\mathtt{x} = \mathtt{e}'$ **in** $\mathtt{e}; \mathtt{x}$. This treatment lets the result of $\mathtt{e}'$ be the final result. Due to the safety guarantee from §5, these two forms will reduce to the same result [6] upon termination.

7. Meta-theory

This section shows the formal properties of $\lambda_{\mathtt{fe}}$. The full proofs could be found in our report [33]. We first show the standard soundness property (Theorem 7.2). Next, we prove that the effect carried in the effect closure is valid (Theorem 7.3), *i.e.*, the passenger expression always has the effect carried by the closure, regardless of how the closure has been passed around or stored. Finally, we present important trace consistency results in Theorem 7.6. Before we proceed, let us define a term that will be used for the rest of the section.

DEFINITION 7.1. *[Redex Configuration] We say* $\langle s; e; f \rangle$ *is a redex configuration of program* $\mathtt{e}'$*, written* $\mathtt{e}' \trianglerighteq <s, e, f>$,

$$\boxed{\text{Definitions:}}$$

$$\begin{array}{lcll}
\mathtt{tc} & ::= & \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle & \textit{effect closure} \\
\mathtt{s} & ::= & \overline{l \rightarrow_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{v}} & \textit{store} \\
f & ::= & \overline{\pi(\rho)} & \textit{trace} \\
\mathtt{v} & ::= & \mathtt{b} \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} \mid l \mid \mathtt{tc} & \textit{value} \\
\mathcal{E} & ::= & - \mid \mathcal{E}\ \mathtt{e} \mid \mathtt{v}\ \mathcal{E} \mid \textbf{let}\ \mathtt{x} = \mathcal{E}\ \textbf{in}\ \mathtt{e} \mid \textbf{ref}\ \rho\ \mathtt{T}\ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := \mathtt{e} \mid \mathtt{v} := \mathcal{E} \mid \textbf{realize}\ \mathcal{E} & \textit{evaluation context} \\
 & \mid & \textbf{if}\ \mathcal{E}\ \textbf{then}\ \mathtt{e}\ \textbf{else}\ \mathtt{e} \mid \textbf{effcase}\ \mathtt{x} = \mathcal{E} : \overline{\mathtt{T}\ \textbf{where}\ P \Rightarrow \mathtt{o}} &
\end{array}$$

$$\boxed{\text{Dynamic Typing:} \quad \mathtt{s}; \Delta; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}$$

$$(\text{ST-LOC})\ \frac{\{l \mapsto_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{v}\} \in \mathtt{s}}{\mathtt{s}; \Delta; \Gamma \vdash_{\mathtt{D}} l : \textbf{Ref}_{\mathtt{r}}\ \mathtt{T}, \emptyset \sim \emptyset} \qquad (\text{ST-TYPE})\ \frac{\mathtt{s}; \Delta; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}{\mathtt{s}; \Delta; \Gamma \vdash_{\mathtt{D}} \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle : \textbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset}$$

For all other (ST-*) rules, each is isomorphic to its counterpart (R-*) rule, except that every occurrence of judgment $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the latter rule should be substituted with $\mathtt{s}; \Delta; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the former.

$$\boxed{\text{Evaluation relation: } \mathtt{s}; \mathtt{e}; f \rightarrow \mathtt{s}'; \mathtt{e}'; f'}$$

$$\begin{array}{lrl}
(\textit{cxt}) & \mathtt{s}; \mathcal{E}[\mathtt{e}]; f \rightarrow \mathtt{s}'; \mathcal{E}[\mathtt{e}']; f, f' & \text{if } \mathtt{s}; \mathtt{e} \rightsquigarrow \mathtt{s}'; \mathtt{e}'; f' \\
(\textit{app}) & \mathtt{s}; \lambda \mathtt{x} : \mathtt{T}.\mathtt{e}\ \mathtt{v} \rightsquigarrow \mathtt{s}; [\mathtt{x} \backslash \mathtt{v}]\mathtt{e}; \emptyset & \\
(\textit{let}) & \mathtt{s}; \textbf{let}\ \mathtt{x} = \mathtt{v}\ \textbf{in}\ \mathtt{e} \rightsquigarrow \mathtt{s}; [\mathtt{x} \backslash \mathtt{v}]\mathtt{e}; \emptyset & \\
(\textit{if}) & \mathtt{s}; \textbf{if}\ \mathtt{b}\ \textbf{then}\ \mathtt{e}_0\ \textbf{else}\ \mathtt{e}_1 \rightsquigarrow \mathtt{s}; \mathtt{e}; \emptyset & \text{if } \mathtt{e} = \mathtt{b}\ ?\ \mathtt{e}_0 : \mathtt{e}_1 \\
(\textit{get}) & \mathtt{s}; !l \rightsquigarrow \mathtt{s}; \mathtt{v}; \textbf{rd}(\mathtt{r}) & \text{if } \{l \mapsto_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{v}\} \in \mathtt{s} \\
(\textit{set}) & \mathtt{s}; l := \mathtt{v} \rightsquigarrow \mathtt{s}, \{l \mapsto_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{v}\}; \mathtt{v}; \textbf{wr}(\mathtt{r}) & \text{if } \{l \mapsto_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{v}'\} \in \mathtt{s} \\
(\textit{ref}) & \mathtt{s}; \textbf{ref}\ \mathtt{r}\ \mathtt{T}\ \mathtt{v} \rightsquigarrow \mathtt{s}, \{l \mapsto_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{v}\}; l; \textbf{init}(\mathtt{r}) & \text{if } l = \textit{freshloc}() \\
(\textit{par}) & \mathtt{s}; \mathtt{e} || \mathtt{e}' \rightsquigarrow \mathtt{s}; \mathtt{e}_0; \emptyset & \text{if } \mathtt{e}_0 = (\mathtt{e}; \mathtt{e}') \text{ or } (\textbf{let}\ \mathtt{x} = \mathtt{e}'\ \textbf{in}\ \mathtt{e}; \mathtt{x}) \\
\hdashline
(\textit{qry}) & \mathtt{s}; \textbf{query}\ \mathtt{e} \rightsquigarrow \mathtt{s}; \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle; \emptyset & \text{if } \mathtt{s}; \emptyset; \emptyset \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \\
(\textit{real}) & \mathtt{s}; \textbf{realize}\ \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle \rightsquigarrow \mathtt{s}; \mathtt{e}; \emptyset & \\
(\textit{effc}) & \mathtt{s}; \textbf{effcase}\ \mathtt{x} = \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle : \overline{\mathtt{T}\ \textbf{where}\ P \Rightarrow \mathtt{e}} \rightsquigarrow \mathtt{s}; \theta \mathtt{e}_i; \emptyset & \text{if } \mathtt{T} <: \theta \mathtt{T}_i \wedge \theta P_i \wedge \theta \mathtt{x} = \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle \\
 & & \wedge \forall j < i, \nexists \theta' \ .\ \mathtt{T} <: \theta' \mathtt{T}_j \wedge \theta' P_j \\
(\textit{rfmt}) & \mathtt{s}; \{\mathtt{T}, \sigma_0 \sim \sigma_1 | P\}\ \mathtt{e} \rightsquigarrow \mathtt{s}; \mathtt{e}; \emptyset & \text{if } s; \emptyset; \emptyset \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \wedge \sigma_0 \subseteq \sigma \wedge \sigma' \subseteq \sigma_1 \wedge P
\end{array}$$

Figure 12. $\lambda_{\mathtt{fe}}$ Operational Semantics.

iff $\vdash e' : T, \sigma \sim \sigma'$, $\emptyset; e'; \emptyset \rightarrow^* s; \mathcal{E}[\mathrm{e}]; f$. *When* e' *is irrelevant, we shorten it as* $\trianglerighteq <s, e, f>$.

THEOREM 7.2 (Type Soundness). *Given an expression* e*, if* $\vdash e : T, \sigma \sim \sigma'$*, then either the evaluation of* e *diverges, or there exist some* s*,* v*, and* f *such that* $\emptyset; e; \emptyset \rightarrow^* s; v; f$.

THEOREM 7.3 (Query-Realize Correspondence). *Given a store* s*, a trace* f*, an expression* $e = \mathcal{E}[\textbf{\textit{query}}\ e']$. *If*

$s;e;f \rightarrow s; \mathcal{E}[\langle e', T, \sigma, \sigma' \rangle]; f \rightarrow^* s'; \mathcal{E}'[\textbf{\textit{realize}}\ \langle e', T, \sigma, \sigma' \rangle]; f'$

then $s; \emptyset; \emptyset \vdash_D e' : T, \sigma \sim \sigma'$*, and* $s'; \emptyset; \emptyset \vdash_D e' : T, \sigma \sim \sigma'$.

To prove trace consistency, we define *trace for expression*:

DEFINITION 7.4 (Trace from Effect Closure). *We say* f *is a trace for expression* e *under store* s*, written* $f \propto <e, s>$*, iff* $s; e; f' \rightarrow^* s'; v; f', f$.

We now define the soundness over traces:

DEFINITION 7.5 (Trace Consistency). *We say* e *is trace-consistent if for any* $\trianglerighteq\ <s, \mathfrak{T}\ e, f'>$*,* $\mathfrak{T} = \{T, \sigma \sim \sigma' | P\}$*, and* $f \propto <e, s>$*, then* $[\sigma \backslash f][\sigma' \backslash f]P$ *holds.*

THEOREM 7.6 ($\lambda_{\mathtt{fe}}$ Trace-Based Consistency). *If* $\vdash e : T, \sigma \sim \sigma'$*, then* e *is trace-consistent.*

8. More Examples

application	**predicate(s)**: $\mathbb{P}$
priority scheduler, §2	<<:
consistent software update, §8.1	#
information security, §8.2	/<<:, <<:
algorithm speculation, §B.2	==
atomicity, §B.3	/<<:
testing, §B.3	<<:

Figure 13. Polarities for Client Predicates.

In this section, we motivate $\lambda_{\mathtt{fe}}$ through a number of applications ranging from custom effect-aware scheduling, to version-consistent dynamic software updating, to data security. We are aimed at demonstrating the benefits of $\lambda_{\mathtt{fe}}$ in two folds. First, it provides flexible and expressive abstractions to address challenging patterns of effect-guided programming. Second, it helps programmers design programs where effect

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Server - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1  let run1 = λprologue, epilogue, update.
2     case epilogue of
3     | [] = realize update
4     | h:t = effcase prologue, epilogue, update:
5            |EC(xl ~ xu), EC(yl ~ yu), EC(zl ~ zu) where zu # xu ∨ zu # yu
6                             =>{prologue # update ∨ epilogue # update} realize update; realize epilogue
7            | default        => h; run1 (query (realize prologue); h) t update in
8   let run = λtransaction, update.
9              run1 (query 0) transaction update
  - - - - - - - - - - - - - - - - - Client - - - - - - - - - - - - - - - - - -
11  let data = ref_u 0 in
12   let fun0 = ref_r λx. x := !x + 1 in
13    let fun1 = ref_w λx. 1 in
14     let update = query (fun0:=(λx. 1);
15                         fun1:=(λx. x:= !x + 1)) in
16      let transaction = query [fun0 data,
17                               fun1 data,
18                               !data] in
19       run transaction update
```

Figure 14. Dynamic Software Updating in First-Class Effects to Preserve Consistency [42].

analysis and manipulation are “correct-by-design,” with refined guarantees specific to individual applications.

The instantiations of the type constraints $\mathbb{P}$ in §3 for the applications in §8 and §B are shown in Figure 13.

8.1 Version-Consistent Dynamic Software Update

Dynamic software update (DSU) [41] allows software to be updated to a new version for software evolution without halting or restarting the software. DSU patches running software with new code on-the-fly. An important property of DSU is *version consistency* (VC) [42]. For VC, programmers specify program points where updates could be applied. Code within two immediate update points is viewed as a *transaction*, *i.e.*, we execute either the old version of the transaction completely or the new version completely.

The listing in Figure 14 defines a piece of `data` and two functions `fun1`, an empty function, and `fun0`, which increments the input by 1. Programmers would like to let a list of three blocks of code in `transaction`, lines 16-18 to be a transaction. The three blocks invoke the functions `fun0` and `fun1` and finally read the `data`. Because one of the functions increases `data` by one, the final result should be `!data + 1`. An example update, lines 14-15, swaps the bodies of the functions `fun0` and `fun1`. One approach is to delay the update until the end of `transaction`, *i.e.*, after the last block `!data` finishes execution. In this case, the transaction executes the old version in the current invocation and will execute the new version in the next invocation.

To increase update availability while ensuring VC, we would like to apply the update when it is available instead of at the end of `transaction`, *e.g.*, the swapping update happens correctly if it is patched after the second block of code `fun1 data`. Here we have executed the two functions whose bodies are to be swapped. The transaction executes the old version completely and the final value of `data` is 1. In contrast, assuming that the updated is patched after `fun0 data`, where we have executed the old version of `fun0` and increased `data` by 1. We will execute the new version of `fun1`, which also increases `data` by 1. The final result is 2, which is unintuitive to programmers.

The second update violates VC, we execute part old code `fun0`, part new code `fun1` and the final result is not correct.

Observe that first-class effects could help reason about whether a patch violates VC at any specific program point. If the effects σ_i of the patch do not conflict ($\#$) with the effects σ_p of code of the transaction before the update, noted as prologue, or effects σ_e of the code after, noted as epilogue, the immediate update (IU) respects VC (details see [42]). In the nutshell, if $\sigma_i \# \sigma_p$, IU is equivalent to applying the update at the beginning of the transaction. On the other hand, if $\sigma_i \# \sigma_e$, IU is equivalent to applying the update at the end of the transaction. This logic of the effects checking is implemented in method `run1` on lines 1-7. It first checks whether the transaction is done, line 3, *i.e.*, the `epilogue` is an empty list `[]`. If so, the `update` could be applied immediately. Otherwise, effect inspection is used to analyze whether the effects of the `update` conflict with both `prologue` and `epilogue`. If there are no conflicts, line 5, update could also be applied at this point. Otherwise, the `update` needs to be delayed.

For example, the effects of the patch on line 15 is writing to the two functions, $\mathbf{wr}_{\mathrm{r}}$, $\mathbf{wr}_{\mathrm{w}}$, the effects of the there blocks of the transaction are below, and ✓ and ✗ represent the effects of the block conflict and do not conflict with the swapping update, respectively:

`!fun0 data`	$\mathbf{rd}_{\mathrm{r}}$, $\mathbf{rd}_{\mathrm{u}}$	✗
`!fun1 data`	$\mathbf{rd}_{\mathrm{w}}$, $\mathbf{rd}_{\mathrm{u}}$	✗
`!data`	$\mathbf{rd}_{\mathrm{u}}$	✓

The update is problematic after the first block, because σ_i conflicts with both σ_p and σ_e, while it is okay after the second block because σ_i only conflicts with σ_p, but not σ_e.

First-class Effects are very well-suited for this application. First, it allows programmers to query the effect of the update, which is important because the update is not available until at runtime. Second, it allows programmers to define their custom conflicting ($\#$) function, such as the ones shown in the *custom conflict model* section in Figure 17, that can go beyond the standard definition “two effects conflict if they access the same memory region” [42], *e.g.*, conflicts on statistical data does not affect the final results of a program and thus could be ignored [19, 40]. Programmers let the type system know this important fact by writing custom effect analyses in predicated pattern matching, line 5. Finally, first-class effects allow programmers to define custom VC correctness criteria, *e.g.*, on line 6, it says the `update` could only be applied if its effects do not conflict with both `prologue` and `epilogue`. This refinement type will be statically verified by $\lambda_{\texttt{fe}}$'s type system.

8.2 Data Zeroing

Information security is of growing importance in applications which interact with third-party libraries, available only at runtime. Consider an example of a bank account in Figure 15. It stores the password in the variable `pw`. It has a method `close`, which will be invoked when a client closes the account. This method accepts a library `x` which displays advertisements when the account is closed [11].

```
 --------------- Bank account ---------------
1 let pw = ref_r 12 in
2  let close = λx.effcase x:
3      |EC(l∼u) where wr_r <<: l =>{wr_r <<: x} realize x
4      |default                  =>pw := -9 in
 -------------- Third party libraries -------------

5   close (query pw := -9);             // Safe Library
6   close (query (if 0 then pw := -9)) // Unsafe
```

Figure 15. Data Zeroing in First-Class Effects Against Leakage of Sensitive Data.

The library could be malicious (*e.g.*, line 6), thus enforcing the security policy that no sensitive data are leaked by the library is vital in protecting the system [11, 35]. We use the zeroing strategy [55]. At runtime, we programmatically analyze the effects of the library and execute it only if it destroys (overwrites) the password in the must-effect $\mathbf{wr}_r$<<: `y`, to avoid the recovery of the original password.

Double-Bounded Effects The must-may effect distinction in $\lambda_{\texttt{fe}}$ is crucial for program correctness. In traditional type-and-effect systems [37, 52], effects are conservative approximations of expressions. This "may-effect" (*i.e.*, over-approximation) may not be expressive enough for a number of applications, including data zeroing. Image the program that would be identical to the one in Figure 15 except that the **effcase** expression where the case on line 3 is predicated by the may-effect, *i.e.*, $\mathbf{wr}_{\texttt{r}}$ <<:`u`. The may-effect view would allow the case to be selected as long as `u` may write to region `r`, such as the problematic client on line 6. Since the may-effect is an over approximation, the evaluation of the expression may not write to `r` at all! As a consequence, the `pw` is not overridden and could be leaked in the future! With double-bounded effects, the distinction is explicit, and programmers use the must-effects on line 3 to ensure that the `pw` must be overridden. We explain how $\lambda_{\texttt{fe}}$ prevents the misuse of the must- and may-effect in §5.1.

The general-purpose zeroing policy is useful in detecting malicious library in many systems, but clients may desire other special-purpose policies, such as the password is not directly read by the library, *a.k.a confidentiality*. In first-class effects, by substituting line 3 with **EC**(`l` ∼ `u`) **where** $\mathbf{wr}_r$ <<:`l` ∧ $\mathbf{rd}_r$ /<<: `u` (`u` does not read `r`), we ensure that the password is not read in the current execution, it is destroyed and thus can not be read in the future, through the combination of confidentiality and zeroing. Other applicable policies are shown in Figure 16.

how	why
zeroing [55]	destroy/overwrite sensitive data
confidentiality [11]	can not read sensitive data
integrity [11]	can not write sensitive data
multiple accesses [49]	can access a subset of data, but not all
negative authorization [5]	can only read non-sensitive data
weak authorization [5]	overridable policy

Figure 16. Representative Information Security Policies in First-Class Effects.

Summary The flexibility of $\lambda_{\texttt{fe}}$ is on par with other security systems and $\lambda_{\texttt{fe}}$ shares the philosophy of improving the flexibility of meta-level designs "hidden" behind effect system [35] by allowing programmers to define custom policy.

8.3 Additional Examples on Scheduling

Thread scheduling is prolific area of research, known to have diverse strategies on schedule ordering, conflict detection, and conflict modeling. In §2, we have already shown the write-before-read strategy. With $\lambda_{\texttt{fe}}$, programmers can flexibly develop a variety of strategies (see Figure 17 for some examples), such as "if the effects of the tasks commute, then execute them concurrently, otherwise sequentially".

In §B, we demonstrate more examples on concurrent programming applications, including efficient algorithmic speculation, cooperative multi-threading, and program testing.

9. Related Work

We are unaware of type-and-effect systems where the (pre-evaluation) effect of an expression is treated as a first-class citizen. The more established route is to treat the post-evaluation effect (in our terms, trace) as a first-class value. In Leory and Pessaux [32], exceptions raised through program execution are available to programmers. This work has influenced many exception handling systems such as Java, where `Exception` objects are also values. Bauer *et al.* [4] extends the first-class exception idea and allows programmers to annotate an arbitrary expression as effect and upon the evaluation of that expression, control is transferred to a matching `catch`-like handling expression as a first-class value. Similar designs also exist in aspect-oriented systems [47]. In general, the work cited here, albeit bearing similar terms, has a distant relationship with our work.

Bañados *et al.* [2] extends the idea of gradual typing [50] to develop a gradual effect system and later Toro *et al.* [53] provides an implementation, which allows programmers

category	scheduling strategy
ordering strategy	FIFO [34, 54]
	LIFO [29]
	random scheduling [27]
	inherit from previous decisions [29]
	write before read [34]
	tasks with less effects first [34]
	tasks with more effects first [34]
	concurrent read, exclusive write [27, 54]
	conflicting tasks in same thread [29, 45]
	task fusion [18, 46]
	divide into no conflicting groups of tasks [46]
	execute conflicting tasks concurrently [9, 10, 19]
	suspend conflicting tasks [56]
conflict detection strategy	latent effect conflict detection [6]
	pairwise tasks conflict detection[27]
	conflict detection with only the last task [34]
	task group conflict detection [54]
custom conflict model	tolerate write/write conflict [19, 40]
	tolerate read/write conflict [19, 40]
	privatization [8, 46, 48]
	speculation [48]

Figure 17. An Example $\lambda_{\texttt{fe}}$ Client Domain: The Menagerie of Scheduling Strategies

customize effect domains. A program element in their system may carry unknown effects, which may become gradually known at runtime. Long *et al.* [35] develops intensional effect polymorphism (IEP), a system that combines dynamic typing and effect polymorphism. Compared with these systems, effect reflection is a first-class programming abstraction in our system, and effect closures are first-class values. This leads to a number of unique contributions we summarized at the end of §1.

There is a large body of work of purely static or dynamic systems for effect reasoning. Examples of the former include Lucassen [37], Talpin *et al.* [52], Marino *et al.* [38], Clarke and Drossopoulou [14], Task Types [28] and DPJ [6]. The latter is exemplified by Soot [30], TWEJava [27], Legion [54], and ATE [34]. Along these lines, Nielson *et al.* [43] is among the most well known foundational system.

The may/must-effect duality may be unique to our system, but double-bounded types is not new. For example, in Java generics, Type bound declarations `super` and `extends` are available for generic type variables [15, 51], the dual bounds in the Java nominal type hierarchy. Unique to our system is that the type checking process actively computes and tightens the bounds of effects — *e.g.*, the type rules of branching and effect analysis — suitable for constructing a precise effect programming and reasoning system. The use of must-effects to enhance expressiveness and in combination with the may-effects to enable non-monotone effect operator in our system may be unique, but type system designers should be able to find conceptual analogies in existing systems, such as liveness [23] and obligation [7].

feature	benefit	reason
effect reflection	precision	employs run-time type information
	flexibility	allows dynamic effect computation
predicated effect analysis	flexibility	allows custom effect analyses
	program correctness	facilitates refinement typing
effect closure	flexibility	implements first-class effects
	soundness	connects expression and effect
refinement type	program correctness	provides refined static guarantees
double-bounded effects	flexibility	provides dual views of effects
	soundness	enforces trace consistency
polarity support	flexibility	enables non-monotone operators
	soundness	enforces trace consistency

Figure 18. A Summary of $\lambda_{\texttt{fe}}$ Features

The `typecase`-style runtime type inspection by Harper and Morrisett [26] and Crary *et al.* [16] is an expressive approach to perform type analyses at runtime. Our **effcase** expression shares the same spirit, except that it works on effect closures. In addition, our pattern matching may be guarded by a predicate, which on the high level can be viewed as an (unrolled/explicit) form of predicate dispatch [20, 39]. In general, supporting expressive pattern matching has a long tradition for data types in functional languages, with many developments in object-oriented languages as well (*e.g.*, [1]). Our work demonstrates how predicated pattern matching interacts with refinement types, dynamic typing, double-bounded effect analysis, and first-class effect support.

Refinement types [13, 22, 25, 44] have received significant attention, with much progress on their expressiveness and decidability. Effect closures in first-class effects are immutable. This language feature significantly simplifies the design of refinement types in $\lambda_{\texttt{fe}}$, as the interaction between refinement types and mutable features is known to be challenging [12]. $\lambda_{\texttt{fe}}$ demonstrates the opportunity of bridging predicated effect analysis and refinement types.

10. Conclusion

We describe a new foundation for effect programming and reasoning, where effects are available as first-class values to programmers. Our system is powered by the subtle interaction among powerful features such as dynamic typing, double-bounded effects, polarity support in predicates, and refinement types. The high-level design features of first-class effects — together with how they are interconnected in an organic fashion — are summarized in Figure 18.

Acknowledgement

This work was partially supported by the US National Science Foundation (NSF) under grants CCF-15-26205, CCF-14-23370, CCF-13-49153, CCF-11-17937, CCF-10-17334, CNS-07-09217, CNS-06-27354, and CAREER awards CCF-08-46059 and CCF-10-54515. We would like to thank John Tang Boyland and reviewers for insightful comments.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the Symposium on Principles of Programming Languages*, 1989.

[2] F. Bañados Schwerter, R. Garcia, and E. Tanter. A theory of gradual effect systems. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2014.

[3] M. Bagherzadeh and H. Rajan. Panini: A concurrent programming model for solving pervasive and oblivious interference. In *the International Conference on Modularity*, 2015.

[4] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *CoRR*, 2012.

[5] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[6] R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *Proceedings of the 25th European Conference on Object-oriented Programming*, 2011.

[7] P. Boström and P. Müller. Modular verification of finite blocking in non-terminating programs. In *Proceedings of the European Conference on Object-oriented Programming*, 2015.

[8] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] J. Burnim, T. Elmas, G. Necula, and K. Sen. Concurrit: Testing concurrent programs with programmable state-space exploration. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, 2012.

[11] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *the Conference on Programming Language Design and Implementation*, 2009.

[12] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *the Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[13] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *the Symposium on Principles of Programming Languages*, 2012.

[14] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2002.

[15] F. Craciun, W.-N. Chin, G. He, and S. Qin. An interval-based inference of variant parametric types. In *the European Symposium on Programming Languages and Systems*, 2009.

[16] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the International Conference on Functional Programming*, 1998.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *the Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[18] R. Dyer. Task fusion: Improving utilization of multi-user clusters. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity*, 2013.

[19] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[20] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *the European Conference on Object-Oriented Programming*, 1998.

[21] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2002.

[22] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.

[23] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[24] A. Greenhouse and J. Boyland. An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*. 1999.

[25] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *the European Conference on Programming Languages and Systems*, 2011.

[26] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, 1995.

[27] S. T. Heumann, V. S. Adve, and S. Wang. The tasks with effects model for safe concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[28] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *the conference on Object-oriented programming, systems, languages, and applications*, 2010.

[29] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 2008.

[30] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *the International Conference on Compiler Construction*, 2005.

[31] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, 2000.

[32] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22, 2000.

[33] Y. Long and H. Rajan. First-Class Effect Reflection for Effect-Guided Programming. Technical Report 16-1, Iowa State U., Computer Sc., 2016.

[34] Y. Long and H. Rajan. A type-and-effect system for asynchronous, typed events. In *Proceedings of the 15th International Conference on Modularity*, 2016.

[35] Y. Long, Y. D. Liu, and H. Rajan. Intensional effect polymorphism. In *Proceedings of the 29th European Conference on Object-oriented Programming*, 2015.

[36] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[37] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

[38] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, 2009.

[39] T. Millstein. Practical predicate dispatch. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2004.

[40] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *the Conference on Programming Language Design and Implementation*, 2007.

[41] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI '06*.

[42] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008.

[43] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, 1999.

[44] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 2008.

[45] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[46] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *the Conference on Programming Language Design and Implementation*, 2011.

[47] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*.

[48] K. Ravichandran and S. Pande. Multiverse: Efficiently supporting distributed high-level speculation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

[49] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3, 2000.

[50] J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[51] D. Smith and R. Cartwright. Java type inference is broken: Can we fix it? In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 2008.

[52] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111, 1994.

[53] M. Toro and É. Tanter. Customizable gradual polymorphic effects for Scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[54] S. Treichler, M. Bauer, and A. Aiken. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

[55] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *the Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[56] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2010.

A. Optimization Dynamic Semantics: Efficient Effect Computation through Pre-computation and Substitution

The operational semantics we have introduced in §6 may not be efficient, because it may require dynamic construction of type derivations to compute dynamic type-and-effects. Here, we introduce one optimization.

Note that the expressions that do not have free variables will have exactly the same static effects (*i.e.*, via static typing) and dynamic effects (*i.e.*, via dynamic typing). Observe that the only difference between the two forms of effects for an expression lies in the typing of the free variables. Because of this, we define an optimized effect computation strategy with two steps:

1. At compile time, the static effects of the expression used by the effect reflection of each **query** expression in the program are computed. Also the type (which contains free type/effect/region variables) of each free variable that appears in the **query** expressio is recorded.
2. At run time, the static types of the free variables are substituted with the dynamic types computed with their corresponding values. This substitution will be used to substitute the effect computed in the previous step to compute the dynamic effect.

For the first step, Figure 19 defines a transformation from the original expression $\mathtt{e}$ to an *annotated expression* $\mathtt{o}$. The two expressions are identical, except that the **query** in the “annotated expression” now takes the form of **query** (

Abstract Syntax in Optimized $\lambda_{\mathtt{fe}}$:

$$
\begin{array}{lcll}
\mathtt{o} & ::= & \mathtt{b} \mid \lambda\mathtt{x}:\mathtt{T}.\mathtt{o} \mid \mathtt{x} \mid \mathtt{o}\ \mathtt{o} \mid \mathbf{let}\ \mathtt{x} = \mathtt{o}\ \mathbf{in}\ \mathtt{o} \mid \mathtt{o}||\mathtt{o} \mid \mathbf{ref}\ \rho\ \mathtt{T}\ \mathtt{o} \mid !\mathtt{o} \mid \mathtt{o} := \mathtt{o} \mid \mathbf{realize}\ \mathtt{o} & \textit{annotated expressions} \\
 & \mid & \mathbf{query}\ (\overline{\mathtt{x}:\mathtt{T}} \rhd \mathtt{T}, \sigma \sim \sigma)\ o \mid \mathbf{if}\ \mathtt{o}\ \mathbf{then}\ \mathtt{o}\ \mathbf{else}\ \mathtt{o} \mid \mathbf{effcase}\ \mathtt{x} = \mathtt{o} : \overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{o}} & \\
\mathtt{oc} & ::= & \langle \mathtt{o}, \mathtt{T}, \sigma, \sigma' \rangle & \textit{effect closure} \\
\mathtt{os} & ::= & \overline{l \rightarrow_{\langle \mathtt{r},\mathtt{T}\rangle} \mathtt{ov}} & \textit{store} \\
\mathtt{ov} & ::= & \mathtt{b} \mid \lambda\mathtt{x}:\mathtt{T}.\mathtt{o} \mid l \mid \mathtt{oc} & \textit{value} \\
\mathcal{O} & ::= & - \mid \mathcal{O}\ \mathtt{o} \mid \mathtt{ov}\ \mathcal{O} \mid \mathbf{let}\ \mathtt{x} = \mathcal{O}\ \mathbf{in}\ \mathtt{o} \mid \mathbf{ref}\ \rho\ \mathtt{T}\ \mathcal{O} \mid !\mathcal{O} \mid \mathcal{O} := \mathtt{o} \mid \mathtt{ov} := \mathcal{O} & \textit{evaluation context} \\
 & \mid & \mathbf{if}\ \mathcal{O}\ \mathbf{then}\ \mathtt{o}\ \mathbf{else}\ \mathtt{o} \mid \mathbf{realize}\ \mathcal{O} \mid \mathbf{effcase}\ \mathtt{x} = \mathcal{O} : \overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{o}} &
\end{array}
$$

Gen function

$$
\begin{array}{rcll}
\mathit{Gen}(\mathbf{Bool}) & = & \mathbf{Bool} & \\
\mathit{Gen}(\mathbf{Ref}_\rho\ \mathtt{T}) & = & \mathbf{Ref}_\rho\ \mathtt{T} & \\
\mathit{Gen}(\mathtt{T} \xrightarrow{\sigma\sim\sigma'} \mathtt{T}') & = & \mathit{Gen}(\mathtt{T}) \xrightarrow{\varsigma\sim\varsigma'} \mathit{Gen}(\mathtt{T}') & \text{where} \quad \varsigma, \varsigma' \text{ fresh} \\
\mathit{Gen}(\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')) & = & \mathbf{EC}(\mathit{Gen}(\mathtt{T}), \varsigma \sim \varsigma') & \text{where} \quad \varsigma, \varsigma' \text{ fresh}
\end{array}
$$

Transformation: $\mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}$

$$
\begin{array}{rll}
\mathbf{query}\ \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathbf{query}\ (\overline{\mathtt{x}:\mathtt{T}} \rhd \mathtt{T}, \sigma \sim \sigma')\ \mathtt{o} & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o} \ \wedge\ \overline{\mathtt{x}} = \mathit{fv}(\mathtt{e}) \ \wedge\ \Gamma(\overline{\mathtt{x}}) = \overline{\mathtt{T}'} \\
 & & \wedge\ \mathit{Gen}(\overline{\mathtt{T}'}) = \overline{\mathtt{T}} \ \wedge\ \Delta;\Gamma, \overline{\mathtt{x} \mapsto \mathtt{T}} \vdash \mathtt{o} : \mathtt{T}, \sigma \sim \sigma' \\
\mathbf{effcase}\ \mathtt{x} = \mathtt{e} : \overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{e}} \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathbf{effcase}\ \mathtt{x} = \mathtt{o} : \overline{\mathtt{T}\ \mathbf{where}\ P \Rightarrow \mathtt{o}} & \text{if } \forall \mathtt{e} \in \overline{\mathtt{e}}.\mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o} \\
\mathbf{realize}\ \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathbf{realize}\ \mathtt{o} & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o} \\
\mathtt{b} \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathtt{b} & \\
\lambda\mathtt{x}:\mathtt{T}.\mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} & \lambda\mathtt{x}:\mathtt{T}.\mathtt{o} & \\
\mathtt{x} \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathtt{x} & \\
\mathbf{ref}\ \rho\ \mathtt{T}\ \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathbf{ref}\ \rho\ \mathtt{T}\ \mathtt{o} & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o} \\
!\mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} & !\mathtt{o} & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o} \\
\mathtt{e} := \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathtt{o} := \mathtt{o}' & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}, \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}' \\
\mathtt{e}\ \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathtt{o}\ \mathtt{o}' & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}, \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}' \\
\mathbf{let}\ \mathtt{x} = \mathtt{e}\ \mathbf{in}\ \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathbf{let}\ \mathtt{x} = \mathtt{o}\ \mathbf{in}\ \mathtt{o}' & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}, \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}' \\
\mathtt{e}||\mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathtt{o}||\mathtt{o}' & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}, \mathtt{e}' \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}' \\
\mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{e}_0\ \mathbf{else}\ \mathtt{e}_1 \overset{\Delta,\Gamma}{\rightsquigarrow} & \mathbf{if}\ \mathtt{o}\ \mathbf{then}\ \mathtt{o}_0\ \mathbf{else}\ \mathtt{o}_1 & \text{if } \mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}, \mathtt{e}_0 \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}_0, \mathtt{e}_1 \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}_1
\end{array}
$$

Evaluation relation: $\mathtt{os}; \mathtt{o}; f \rightarrow_O \mathtt{os}'; \mathtt{o}'; f'$

$$
\begin{array}{llcll}
(\mathit{Ocxt}) & \mathtt{os}; \mathcal{O}[\mathtt{o}]; f & \rightarrow_O & \mathtt{os}'; \mathcal{O}[\mathtt{o}']; f, f' & \text{if} \quad \mathtt{os}; \mathtt{o} \rightsquigarrow_O \mathtt{os}'; \mathtt{o}'; f' \\
(\mathit{Oqry}) & \mathtt{os}; \mathbf{query}\ (\overline{\mathtt{ov}:\mathtt{T}} \rhd \mathtt{T}, \sigma \sim \sigma')\ o & \rightsquigarrow_O & \mathtt{os}; \langle \mathtt{o}, \theta\mathtt{T}, \theta\sigma, \theta\sigma' \rangle ; \emptyset & \text{if} \quad \mathtt{os}; \emptyset; \emptyset \vdash_{\mathtt{D}} \overline{\mathtt{ov}} : \overline{\mathtt{T}'}, \emptyset \sim \emptyset \ \wedge\ \theta\overline{\mathtt{T}} = \overline{\mathtt{T}'}
\end{array}
$$

For all other $\rightsquigarrow_O$ rules, each is isomorphic to its counterpart $\rightsquigarrow$ rule, except that every occurrence of metavariable $\mathtt{e}$, $\mathtt{tc}$, $\mathtt{s}$, $\mathtt{v}$, and $\mathcal{E}$ in the latter rule should be substituted with $\mathtt{o}$, $\mathtt{dtc}$, $\mathtt{os}$, $\mathtt{ov}$, and $\mathcal{O}$ in the former.

Figure 19. Optimized $\lambda_{\mathtt{fe}}$.

$\overline{\mathtt{x}:\mathtt{T}} \rhd \mathtt{T}, \sigma \sim \sigma')\ \mathtt{o}$, which records the free variables of expressions $\mathtt{o}$ and their corresponding static types, denoted as $\overline{\mathtt{x}:\mathtt{T}}$. The same expression also records the statically computed type $\mathtt{T}$, must-effects σ and may-effects σ' for $\mathtt{o}$. The function $\mathit{fv}(\mathtt{o})$ computes the free variables for $\mathtt{o}$.

Considering all the annotated information is readily available while we perform static typing of the **query** expression — as in (T-QUERY) — the transformation from expression $\mathtt{e}$ to annotated expression $\mathtt{o}$ under Δ and Γ, denoted as $\mathtt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \mathtt{o}$, is rather predictable, defined also in Figure 19.

The reduction system $\rightarrow_O$ is defined at the bottom of the same figure and the notation $\rightarrow_O^*$ represents the reflexive and transitive closure of $\rightarrow_O$. Upon the evaluation of the annotated **query** expression, the types associated with the free

```
- - - - - - - - - - - - - - - - - - Tasks with Effect - - - - - - - - -
1 let s = λ fn.
2  let t1 = query fn x1 in
3   let t2 = query fn x2 in
4    let t3 = query fn x3 in
5     effcase t1, t2, t3:
6     |EC(T0,y4 ~ y0), EC(T1,y5 ~ y1), EC(T2,y6 ~ y2)
7       where (y0 # y1) ∧ (y0 # y2) ∧ (y1 # y2)
8             =>realize t1||realize t2||realize t3
9     |default=> fn x1; fn x2; fn x3 in
```

```
- - - - - - - - - - - - - - - - Latent Effect Analysis - - - - - - -
10 let s = λ fn.
11  let t = query fn in
12
13
14   effcase t:
15   |EC(T0 --y0~y1--> T1,y ~ z)
16      where (y1 # y1)
17            =>realize t x1||realize t x2||realize t x3
18   |default => fn x1; fn x2; fn x3 in
```

Figure 20. SIMD in First-Class Effects. On the Left, the Effects of each Pair of the Tasks Are Checked to Verify Concurrency Safety (line 7), O(n^2) Complexity. On the Right, Latent Effects Are Checked (line 16), O(n) Complexity.

variables — now substituted with values — are substituted with the types associated with the corresponding values. The latter is computed by judgment $\texttt{os};\Delta;\Gamma \vdash_{\mathrm{D}} \texttt{o} : \texttt{T},\sigma \sim \sigma'$, defined as $\texttt{s};\Delta;\Gamma \vdash_{\mathrm{D}} \texttt{e} : \texttt{T},\sigma \sim \sigma'$ where $\texttt{e} \overset{\Delta,\Gamma}{\rightsquigarrow} \texttt{o}$, substituted types of the free variables with the types of the values.

```
1 let buf = ref_r -1 in
2  let speculate =
3    λx.effcase x:
4       |EC(wr_r~wr_r)=>{x == wr_r} realize x||realize x
5       |default      =>/* default copying strategy */ in
6   speculate (query buf := 1)
```

Figure 21. Algorithmic Speculation [48] in First-Class Effects to Improve Performance.

B. More Applicability

B.1 Latent Effect Analysis

With $\lambda_{\texttt{fe}}$, programmers can inspect the structure of effects, allowing latent effect associated with higher-order function to be analyzed or more generally any effects nested in the types. To illustrate, consider the example in Figure 20. Assume that we have already defined three references bf0, bf1, and bf2 in different regions r, u, and w respectively. The funciton s accepts a function fn as an argument and decides whether it can be applied on the three references in parallel, so called *single instruction multiple data* (SIMD) application. With the **effcase** expression, programmers can inspect the latent effects of the instruction to verify the concurrency safety [6] (line 16). In contrast, traditional systems [27, 35] will create a task for each data, and check that the effects of each pair of tasks do not interfere (lines 2-7), that can lead to $O(n^2)$ checks, where n is the size of the data. In a similar vein, such a design eases the effects reasoning of the *map*, *filter*, *select* functions in the *MapReduce* and *ParallelArray* framework [17, 31].

B.2 Algorithmic Speculation

We highlight the use of the non-monotone operator == (line 4), in Figure 21, which is not supported or can be hard to reason about in previous system [35]. The operator == is satisfied if the effects of LHS and RHS are equal.

Typically, in algorithmic speculation [48], a master process executes a computation in multiple worker processes and returns with the fastest among them. The runtime will copy the heap from the master to the worker whenever the worker reads the heap and copy the heap from the best worker back to the master upon completion. The workers are organized into topology to reduce copying overhead. With $\lambda_{\texttt{fe}}$, overhead can be further reduced.

If the to-be-speculated computation x is pure and only writes the final result to the returning buf, then the copying overhead can be avoided. The programmer analyzes its effects (line 4) and if both its must- and may-effects are $\mathbf{wr}_r$, *i.e.*, $\textsc{ec}(\mathbf{wr}_r \sim \mathbf{wr}_r)$, x can be evaluated in two threads and the result from the faster one is used. As such, expensive copying by executing them in processes is avoided.

B.3 Atomic Cooperative Multi-threading and Testing

Writing concurrent programs is difficult, error prone, and creates code which is hard to debug. Consider an adapted real world bug shown in Figure 22. It is intuitive that in the function buggy, if the value of buf is not 0 on line 8, then its value will not be 0 on the immediate next line either. However, such an intuition could be violated by the expression on line 10 from the interfere thread during concurrent execution, causing a program crash [36].

The root cause of the problem is that programmers usually think sequentially and assume that a small block of code will be executed atomically, which is not provided in the preemptive semantics [3, 56]. Under the preemptive semantics, an yield expression will be inserted into the program points, wherever references are accessed, *e.g.*, on line 9. The yield expression serves as a breakpoint to pause the current thread, from which control could be transferred to other threads, *e.g.*, the interfere thread.

Testing Previous work [10] on programmer-guided testing has proposed to let programmers to decide which thread the control will be transferred to in order to trigger bug faster, within the yield function.

With first-class effects, this can be done in a more effective matter. By inspecting the effect, the programmer schedules threads that must write to region r (*e.g.*, line 3). This technique is arguably more effective because priority is given to conflicting threads, which may trigger bugs. By

```
---------------- Yield in Testing -------------
1 let yield = λx. for y : !threads
2     effcase y:
3     |EC(h ~ z) where wr_r <<: h =>{wr_r <<: y} realize y
```

```
---------------- Yield in Atomicity ---------------
4 let yield = λx. for y : !threads
5     effcase y:
6     |EC(h ~ z) where wr_r /<<: z =>{wr_r /<<: y} realize y
```

```
------------------------------------- Application code -------------------------------------
7  let buf = ref_r 0 in
8   let buggy = λx. if !buf not 0
9            then yield buf; /* break point */  10 / !buf in         // potential divide by 0 error
10   let interfere = λ x. !buf := 0 in
11    let threads = ref (cons (query buggy 0) (query interfere 0)) in
12     buggy || interfere
```

Figure 22. Programmer-Guided Testing and Atomicity Preservation in First-Class Effects, Inspired by Yi *et al.* [56], Burckhardt *et al.* [9], Erickson *et al.* [19] and Burnim *et al.*'s Work [10].

```
λ x1, x2, x3.
 effcase x1, x2, x3:
 |EC(y4~y0), EC(y5 ~ y1), EC(y6 ~ y2) where y0 # y1 ∧ y0 # y2 ∧ y1 # y2=>realize x1||realize x2||realize x3
 |EC(y4~y0), EC(y5 ~ y1), EC(y6 ~ y2) where y0 # y1                     =>(realize x1||realize x2); realize x3
 |EC(y4~y0), EC(y5 ~ y1), EC(y6 ~ y2) where y0 # y2                     =>(realize x1||realize x3); realize x2
 |EC(y4~y0), EC(y5 ~ y1), EC(y6 ~ y2) where y1 # y2                     =>(realize x2||realize x3); realize x1
 |default                                                               =>realize x1; realize x2; realize x3
```

Figure 23. Don't-care Non-determinism in First-Class Effects. Expression $\mathrm{e} \parallel \mathrm{e}'$ Denotes Running e and e' in Parallel. Binary Operator # Is Satisfied If the Effects of LHS and RHS Are Disjoint.

specifying the must-effect, threads that must write to `r` will be executed and the *precision* can be enhanced. Alternatively, if the predicate on line 3 is substituted with **where** $\mathbf{wr}_r$ <<: h =>, *recall* will be enhanced. Interestingly, programmers can choose to increase precision or recall by choosing the must- and may-effects. Note that the effects of the threads are retrieved once when the threads are created (line 11) and may be reused many times.

Maintaining Atomicity Conversely, for the application of atomicity, programmers aim at achieving concurrency safety, instead of triggering bug. The programmer-defined atomic scheduler analyzes the effects of the threads, and allows only the threads that do not write `r` to be run in concurrent, line 6, *i.e.*, pausing all conflicting threads.

B.4 Application Specific Effect Analyses

one man's meat is another man's poison

To increase flexibility, $\lambda_{\texttt{fe}}$ allows programmers to decide which expression to query and to customize their effect analyses to meet their domain knowledge. The value of domain knowledge can be viewed in the application for safe parallelism. The `fork` function would like to execute the three tasks `x1`, `x2`, and `x3` in parallel. Let the predefined ternary operator **spawn** denote running the tasks in parallel if their effects are pairwise disjoint and sequentially otherwise.

```
let fork = λ x1, x2, x3.
                 spawn x1, x2, x3 in ...
```

Clearly, the `fork` function maintains sequential consistency, *i.e.*, the potential concurrent execution of the tasks is *behaviorally equivalent* to executing the tasks one by one. This property is the most intuitive model for sequentially-trained programmers to understand and reason about their programs [36]. However, such property is neither sufficient nor necessary for several applications.

It is not sufficient (correctness), *e.g.*, certain applications [34] will reorder the conflicting tasks such that producer tasks are executed before comsumer tasks, to ensure proper initialization or to prioritize producer tasks.

It is not necessary (performance), *e.g.*, several effect systems [27, 34, 54] will reorder the tasks and execute non-conflicting tasks in parallel. so called the *don't-care non-determinism* parallelism.

```
fork reader writer reader
```

For example, for the above call site of, an user may wish to run the two `readers` concurrently, and after they are done, run `writer`, but `fork` will run them sequentially. Such concurrent execution may have huge performance benefits compared with the sequential execution. Paradoxically, the reordering is only correct in certain domain but not the others [46], which certainly requires domain knowledge.

Other effect-aware schedulings are possible and are shown in Figure 17. While, previous work allows programmers to select a small predefined subset of schedulings, in $\lambda_{\texttt{fe}}$, programmeres are endowed the power of implementing any of the schedulings and combining them in any order.

With $\lambda_{\texttt{fe}}$, programmers can implement don't-care non-determinism scheduling as in Figure 23.