First-Class Effect Reflection for Effect-Guided Programming

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

\textsuperscript{\alpha,\gamma}Iowa State University, USA \quad \textsuperscript{\beta}SUNY Binghamton, USA

\texttt{\{csgzlong,hridesh\}@iastate.edu \quad \texttt{davidl@cs.binghamton.edu}}

Abstract

This paper introduces a novel type-and-effect calculus, \textit{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 “hard-coded” 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 \textit{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, 53], 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 \textit{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, $\lambda_{fe}$, is endowed with powerful programming abstractions:

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

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

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

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 $\lambda_{fe}$ 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 $\lambda_{fe}$, programmers can flexibly develop a variety of thread management strategies. We will further demonstrate a broad range of applications beyond thread scheduling in \S\ref{sec:ex} and \S\ref{sec:discussion}. 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.\textsuperscript{1} The grand challenge of designing a flexible programming model lies in

\textsuperscript{1} Marvel’s Spiderman
principled and precise reasoning. The core technical development of $\lambda_{te}$ 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 $\lambda_{te}$ that employs run-time information to improve precision. Second, $\lambda_{te}$ provides static guarantees to application-specific effect management properties through refinement types, promoting “correct-by-design” effect-guided programming. Third, $\lambda_{te}$ 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 $\mu$ vs. obligation $\nu$ 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 $\lambda_{te}$.

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 $\lambda_{te}$ 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, $\lambda_{te}$ 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 $\lambda_{te}$ 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 $x$) first, given the two input tasks $x_1$ and $x_2$. 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 $\lambda_{te}$, 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 5 or $scheduler$ $writer$ $reader$.

In $\lambda_{te}$, 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 3 the case is matched if the $must$-effect of $x_1$ writes to region $r$, noted $wr$, $<<:1$. Effects in $\lambda_{te}$ have both lower bound and upper bounds, e.g., $x_1$ has $must$-effect $1$ and $may$-effect $u$, whose type is noted as $EC(l \sim u)$. The effect is realized on line 6 leading to the evaluation of the generator expression. Finally, to guarantee that the write-before-read ordering is correctly implemented, the scheduler uses the refinement type $\{wr, <<:c \mid \forall wr, <<:p\}$, 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 $x_1$ and $x_2$ 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 \( x_1 \) is a superset of the effect \( \text{wr}_r \), then execute \( x_1 \) first. Intuitively, what the programmer indeed means is that the trace — informally, the “post-evaluation effect” — of \( x_1 \) is a superset of \( \text{wr}_r \). 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 \( x_1 \) is executed first when the traces do not conform to presupsetting 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, \( \lambda_{\text{fe}} \) 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, \( \lambda_{\text{fe}} \) 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 \( r \circ \)” and must-effect \( \emptyset \), are likely to be computed by a purely static effect system.

```
let client = λ buf,
  let w = (query buf := 1) in
  effcase w:
  | EC(l1 ~ u1) where wr_r, <=; l1 => realize w
  | client (if 1 > 0 then ref_0 else ref_0 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.

```
client = λ buf,
  let w = (query buf := 1) in
  effcase w:
  | EC(l1 ~ u1) where wr_r, <=; l1 => realize w
  | client (if 1 > 0 then ref_0 else ref_0 0)
```

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 \( \times \) 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 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 45.

### 2.5 Trace-Effect Correspondence

To meet the fourth challenge, we define the notion of trace consistency. The crucial question in \( \lambda_{\text{fe}} \) 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 \( \lambda_{\text{fe}} \), 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 \( \{ \text{wr}_r, <=; c \)
\[ \mapsto \] \text{wr}_r \hspace{1em} \triangleq \hspace{1em} \{ p \} \text{, type checks, and the memory traces of evaluating } c \text{ and } p \text{ are } f_c \text{ and } f_p \text{ respectively, then if } f_c \text{ contains a write to } r, \text{ then } f_p \text{ 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 \( \lambda_{fe} \), 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 \( \text{wr}_r \triangleq 1 \). The may-effects of the reader is \( \text{wr}_r \), thus the case on line 4 is matched and the reader is evaluated first.

1. let \( \text{buf} \) = \( \text{ref} \), 0 in
2. let \( \text{scheduler} \) = \( \lambda \) \( x_1 \), \( x_2 \).
3. let \( (p, c) \rightarrow \text{effcase } x \triangleq e : T \) where \( P \Rightarrow e \).
4. (\( \text{wr} \triangleq c \rightarrow \text{wr}_r \triangleq p \) ) \text{ realize } e \text{ in } c \text{ in }
5. let \( \text{reader} \) = \( (\text{query if } 0 < 1 \text{ then } \text{buf} \Rightarrow 1 \) in
6. let \( \text{writer} \) = \( (\text{query buf} \Rightarrow 1) \) in
7. scheduler \text{ 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 \( \triangleq \), \( \lambda_{fe} \) assigns the RHS of \( \triangleq \) a + polarity to indicate that a must-effect should be used and the RHS a −. Examples for other polarities, such as − and i, are shown in the table below.

<table>
<thead>
<tr>
<th>polarity</th>
<th>name</th>
<th>example</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>monotone decreasing</td>
<td>LHS of ( \triangleq )</td>
<td>may-effect</td>
</tr>
<tr>
<td>+</td>
<td>monotone increasing</td>
<td>RHS of ( \triangleq )</td>
<td>must-effect</td>
</tr>
<tr>
<td>i</td>
<td>invariant</td>
<td>( \Rightarrow )</td>
<td>may equals must</td>
</tr>
</tbody>
</table>

**Figure 2.** Polaris 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 \( \text{wr}_r \triangleq \). Intuitively, the must-effect \( 1 \) is a subset of the trace \( f_{x_1} \) and \( f_{x_1} \) is a subset of the may-effect \( u \). Therefore, if this case is matched, \( \text{wr}_r \) is a subset of \( f_{x_1} \), i.e., \( x_1 \) 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 §9.

### 3. \( \lambda_{fe} \): a Calculus with First-Class Effects

The abstract syntax of \( \lambda_{fe} \) 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 \( \lambda \) 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 \( e |\triangleright| e \). 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 \( \text{let x = e in e}' \).

**Effect management.** consists of the key abstractions described in §2. Expression \( \text{query e} \) dynamically computes the effect of expression \( e \). The result of a query is an effect closure. Programmers can inspect a closure with \( \text{effcase x = } e : T \hspace{1em} \text{where } P \Rightarrow e \). 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 \( \text{realize e} \) 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, \( e.g. \), \( \text{effcase x = x_0 } : \hspace{1em} T \hspace{1em} \text{where } P \Rightarrow e \), we shorten it to \( \text{effcase x_0 } : \hspace{1em} T \hspace{1em} \text{where } P \Rightarrow e \), as is the case in the examples.

This effect analysis expression pattern-matches the closure against the type patterns \( T \). \( P \) is a type constraint to further refine pattern matching. It supports connectives of proposition logic, together with the atomic \( n \)-ary form \( P \sigma \), left abstract, which can be concretized into different forms for different concrete languages. Please find examples of different instantiations of \( P \) in §8.

**Effects and Effect Types.** Effects are region accesses and have the form \( \pi \rho \), representing an access right \( \pi \) to values in region \( \rho \). Access rights include allocation \( \text{init} \), read \( \text{rd} \) and write \( \text{wr} \).
Compared with existing effect systems, our system supports both must- and may-effects. Function types $T \overset{\sigma \sim \sigma'}{\rightarrow} T'$ specifies a function from $T$ to $T'$ with must-effect $\sigma$ and may-effect $\sigma'$ as the effects of the function body.

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

Regions. The domain of regions is the disjoint union of a set of constants $\pi$. The region abstracts memory locations in which it will be allocated at runtime. Our notion of region is standard [57, 52]. In $\lambda_{\pi}$, allocation sites are explicitly labelled with regions. Region inference is feasible [21, 24], an issue orthogonal to our interest. In $\lambda_{\pi}$, type $\alpha$, effect $\zeta$, and region $\gamma$ variables are cumulatively referred to as “pattern variables”, and we use a metavariable $g$ for them. A type variable $\alpha$ can be used in the effcase expressions to match any type $T$, given that the constraint $P$ is satisfied if $\alpha$ is substituted with $T$, 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 $\text{dom}$ and $\text{rng}$ are the conventional and range functions. Substitution $\theta$ maps type variables $\alpha$ to types $T$, region variables $\gamma$ to regions $\rho$, and effect variables $\zeta$ to effects $\sigma$. 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 $T <: T'$ says $T$ is a subtype of $T'$ 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 $\sigma$, it is always OK to be provided with a function that must produce a “superset effect” of $\sigma$. On the flip side, for a program point that expects a function that may produce effect $\sigma$, it is always OK to be provided with a function that may produce a “subset effect” of $\sigma$. As expected, effect subsumption in our system — the “superset effect” and the “subset effect” — is supported through set containment over $\sigma$ elements. Effect closure types follow a similar design, as seen in (sub-EC).

<table>
<thead>
<tr>
<th>Subtyping: $T &lt;: T'$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(sub-REFL)</td>
<td>$T &lt;: T$</td>
<td>$T' &lt;: T'$</td>
</tr>
<tr>
<td>(sub-TRANS)</td>
<td>$T &lt;: T''$</td>
<td>$T' &lt;: T''$</td>
</tr>
<tr>
<td>(sub-REF)</td>
<td>$\rho \subseteq \rho'$</td>
<td>$\text{Ref}<em>{\rho} T &lt;: \text{Ref}</em>{\rho'} T$</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sub-FUN}$</td>
<td>$T'<em>{x} &lt;: T</em>{x}$</td>
<td>$T &lt;: T'$</td>
</tr>
<tr>
<td></td>
<td>$T''<em>{x} &lt;: T''</em>{x}$</td>
<td>$\sigma_{2} \subseteq \sigma_{0}$</td>
</tr>
<tr>
<td></td>
<td>$T'<em>{x} \triangleright \sim \sigma</em>{3}$</td>
<td>$\sigma_{1} \subseteq \sigma_{3}$</td>
</tr>
</tbody>
</table>

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 $\Gamma$ maps variables to types:

$$\Gamma ::= \forall x : T$$

Notation $\Gamma(x)$ denotes $T$ if the rightmost occurrence of $x : T$ in $\Gamma$ is $x : T$.

Type checking is defined through judgment $\Gamma \vdash e : T$, $\sigma \sim \sigma'$, defined in Figure 5. The judgment says under type environment $\Gamma$, expression $e$ has type $T$, must-effect $\sigma$ and may-effect $\sigma'$. 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. In effect, given an expression as follows and that val is in region $r$, the must- and may-effects are $\{rd_{r}, wr_{r}\}$ respectively:

$$\text{if } x > 0 \text{ then } !\text{val} \text{ else } \text{val} := !\text{val} + 1$$

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
**Type Checking:** $\Gamma \vdash e : T, \sigma \sim \sigma'$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T-EFFCASE)</td>
<td>$\Gamma \vdash e : \text{EC}(\tau, \sigma \sim \sigma'), \emptyset \sim \emptyset \quad \exists \xi, T_n \equiv \xi \land P_n = \text{true} \quad \Gamma, x \vdash \text{EC}(\tau, \sigma_i \sim \sigma_j') \vdash e_1 : T, \sigma_i'' \sim \sigma_j'''$, for all $i \in {1 \ldots n}$</td>
</tr>
<tr>
<td>(T-QUERY)</td>
<td>$\Gamma \vdash e : T, \sigma \sim \sigma' \quad \Gamma \vdash \text{query } e : \text{EC}(\tau, \sigma \sim \sigma'), \emptyset \sim \emptyset$</td>
</tr>
<tr>
<td>(T-REALIZE)</td>
<td>$\Gamma \vdash e : \text{EC}(\tau, \sigma_0 \sim \sigma_1), \sigma_2 \sim \sigma_3 \quad \Gamma \vdash \text{realize } e : T, \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3$</td>
</tr>
<tr>
<td>(T-ABS)</td>
<td>$\Gamma, x \mapsto T \vdash e : T', \sigma \sim \sigma' \quad \Gamma \vdash \lambda \chi : T. e : T \xrightarrow{\sigma \sim \sigma'} T', \emptyset \sim \emptyset$</td>
</tr>
<tr>
<td>(T-APP)</td>
<td>$\Gamma \vdash e : T, \sigma \sim \sigma' \quad \Gamma \vdash e' : T', \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3 \cup \sigma_5$</td>
</tr>
<tr>
<td>(T-LET)</td>
<td>$\Gamma \vdash e : T, \sigma \sim \sigma_1 \quad \Gamma, x \mapsto T \vdash e' : T', \sigma_2 \sim \sigma_3 \quad \Gamma \vdash \text{let } x = e \Rightarrow e' : T', \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3$</td>
</tr>
<tr>
<td>(T-REF)</td>
<td>$\Gamma \vdash e : T, \sigma \sim \sigma' \quad \Gamma \vdash \text{ref } \rho \vdash e : \text{Ref}<em>\rho, T, \sigma \cup \text{init}</em>\rho \sim \sigma' \cup \text{init}_\rho$</td>
</tr>
<tr>
<td>(T-SET)</td>
<td>$\Gamma \vdash e : \text{Ref}<em>\rho, \sigma_0 \sim \sigma_1 \quad \Gamma \vdash e' : T, \sigma_0 \cup \sigma_2 \cup \text{wr}</em>\rho \sim \sigma_1 \cup \sigma_3 \cup \text{wr}_\rho \quad \Gamma (x) = T$</td>
</tr>
<tr>
<td>(T-IF)</td>
<td>$\Gamma \vdash \text{if } e \text{ then } e_0 \text{ else } e_1 : T, \sigma_0 \cup (\sigma_2 \cup \sigma_4) \sim \sigma_1 \cup \sigma_3 \cup \sigma_5$</td>
</tr>
</tbody>
</table>

**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, $\lambda x e$ 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 4 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 $\exists \chi$ takes the form of $\{ T, \sigma \sim \sigma' \mid P \}$, where predicate $P$ is used to refine the base type $T$ and effects $\sigma \sim \sigma'$, a common notation in refinement type systems [12] [25] [44]. When $T$, $\sigma$ and $\sigma'$ 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 (wr, $\ll < < > \ll \rightarrow \ll p$). We extend the subtyping relation with one additional rule, (subr-REFINE). Here, subtyping of refinement types is defined as the logical implication $\rightarrow$ of the predicates of the two types.
For all other (subr-REFINE) rules, each is isomorphic to its counterpart (subr-* ) rule in Figure 4.

5.1 Polarity Support

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

(bivariance) $\ast \leftarrow \sqcup \uparrow \varsigma$

(covariance) $- \leftarrow \sqcup \uparrow i$

(invariance) $i \leftarrow \sqcup \uparrow :$

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 $\sigma$ always implies a predicate application identical with the former except argument $j$ being $\sigma'$ and $\sigma \subseteq \sigma'$, shown in (mono-$\Rightarrow$) in Figure 9 where the $\mathcal{V}(P, j)$ (Figure 8) notation gets the $j$th polarity of the predicate $P$, e.g., $\mathcal{V}(<<; 0)$ will return the polarity of LHS of $<<; i.e., and $\mathcal{V}(<<; 1)$ 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 $\sigma$ always implies a predicate application identical with the former except argument $j$ being $\sigma'$ and $\sigma' \subseteq \sigma$, as (mono-$\Downarrow$). 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 3. 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, -1 in
2. let fun = \x:exact, y:exact. effcase x, y:
3. | EC(l0 ~ l0), EC(l1 ~ l1)
4. where l0 = l1 => (y == x) \& y in
5. fun (query !buf) (query !buf);
6. fun (query query buf := 0) (query !buf)

<table>
<thead>
<tr>
<th>$\mathcal{P}$</th>
<th>$LHS(j = 0)$</th>
<th>$RHS(j = 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;$</td>
<td>-(may)</td>
<td>+(must)</td>
</tr>
<tr>
<td>$&lt;=$</td>
<td>+(must)</td>
<td>-(may)</td>
</tr>
<tr>
<td>$#$</td>
<td>-(may)</td>
<td>-(may)</td>
</tr>
<tr>
<td>$==$</td>
<td>i(may and must)</td>
<td>i(may and must)</td>
</tr>
</tbody>
</table>

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

Figure 9. Predicate Implication $\Rightarrow$.

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 $x==$ and that both $x$ and $y$ have tight bounds, their traces must be equal.

The must-effect of the expression on line 7 is $\{rd_x\}$, the may-effect is $\{rd_x, wr_r\}$, 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 5-8 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 1. Similarly, the **exact** annotation fulfills a similar task in Figure 1. Without **exact**, the program will not be sound.
Refinement Type Checking: $\Delta; \Gamma \vdash e : T, \sigma \sim \sigma'$

(R-REFINE) $\frac{\Delta \vdash P \quad \Delta; \Gamma \vdash e : T, \sigma \sim \sigma' \quad [\Gamma] \rightarrow P}{\Delta; \Gamma \vdash \{T, \sigma \sim \sigma'|P\} : e : T, \sigma \sim \sigma'}$

$\Delta; \Gamma \vdash e : \text{ec}(T, \sigma \sim \sigma'), \emptyset \sim \emptyset \quad \exists \xi. \text{ec}(T_i, \sigma_i \sim \sigma_i') < : \text{ec}(T, \sigma \sim \sigma'), \text{for all } i \in \{1, \ldots, n\}$

$\exists \xi. T_n = \xi \land P_n = \emptyset \quad \Delta_i = \Delta \sqcup \text{polar}_i(T_i) \sqcup \text{polar}_e(\sigma_i \sim \sigma_i')$

(R-EFFCASE) $\frac{\Delta_i \vdash P_i \quad \Delta_i; \Gamma \vdash x \equiv \text{ec}(T_i, \sigma_i \sim \sigma_i'), \emptyset \sim \emptyset | P_i \vdash e_i : T_i, \sigma_i'' \sim \sigma_i''' \sim \sigma_i', \text{for all } i \in \{1, \ldots, n\}}{\Delta; \Gamma \vdash \text{effcase } x = e : \text{ec}(T, \sigma \sim \sigma') \text{ where } P \Rightarrow e : T_i \cap \exists \zeta \in \{1, \ldots, n\} \sigma_i'' \sim \zeta \in \{1, \ldots, n\} \sigma_i'''}$

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

**Type Checking Predicate:** $\Delta \vdash P$

<table>
<thead>
<tr>
<th>$\neg \Delta \vdash P$</th>
<th>$\Delta \vdash P$</th>
<th>$\Delta \vdash P'$</th>
<th>$\Delta \vdash P \wedge P'$</th>
<th>$\forall \sigma_j \in \sigma \in \sigma_j \text{ s.t. } \zeta \in \text{dom}(\Delta), \Delta(\zeta) \vdash \text{ec}(P, j)$</th>
<th>$\Delta \vdash P \sigma$</th>
</tr>
</thead>
</table>

**Predicate Combination:** $[\Gamma] = P$

$[\xi \mapsto \{T, \sigma \sim \sigma'|P\}] = \bigwedge_{i=1}^{n} P_i$

**Environment Negation:** $\neg \Delta = \Delta$

$\neg \xi \mapsto \gamma' = \xi \mapsto \neg \gamma'$

**Polarity Negation:** $\neg \gamma' = \gamma'$

$\neg j = +$

$\neg j = +$

$\neg j = *$

$\neg j = i$

**Computing $\Delta$ from Type:** $\text{polar}_i(T) = \Delta$

$\text{polar}_i(\text{Bool}) = \emptyset$

$\text{polar}_i(\alpha) = \emptyset$

$\text{polar}_i(\text{Ref}, x) = \text{polar}_i(T)$

$\text{polar}_i(T, \sigma \sim \sigma') = \text{polar}_i(T) \sqcup \text{polar}_i(T') \sqcup \text{polar}_i(\sigma \sim \sigma')$

$\text{polar}_i(\text{ec}(T, \sigma \sim \sigma')) = \text{polar}_i(T) \sqcup \text{polar}_i(\sigma \sim \sigma')$

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

$\text{polar}_i(\zeta \sim \zeta) = \zeta \mapsto i$

$\text{polar}_i(\pi \sim \pi') = \emptyset$

$\text{polar}_i(\sigma \cup \sigma) = \zeta \mapsto + \sqcup \text{polar}_i(\sigma \sim \sigma')$

$\text{polar}_i(\sigma \sim \sigma \cup \sigma') = \zeta \mapsto - \sqcup \text{polar}_i(\sigma \sim \sigma')$

**Figure 10.** Typing Rules for Checking Refinement Types.

**Figure 11.** Functions for Computing Effect Polarity.

In this presentation, our polarity modifiers are coarsely-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**<sub>&rarr;</sub> 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 10 can be updated where every occurrence of **exact** can be replaced by **exact**<sub>&rarr;</sub>. We here do not formalize this useful but predictable extension.

### 5.2 Refinement Type Checking

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

The differences between (R-EFFCASE) and (T-EFFCASE) (in Figure 11) 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 $\zeta$ appearing in the must-effect, will have $+$ polarity. If $\zeta$ appears in the may-effect, it will have $-$ polarity. If the must- and may-effects are the same, $\zeta$, $\zeta$ has $i$ polarity, otherwise $\zeta$ 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 $\zeta$ 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 $\ll$; and for its negation $\ll\ll$, 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 $x$ 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 $\lambda_{te}$ are immutable. This language feature significantly simplifies the design of refinement types in $\lambda_{te}$, as the interaction between refinement types and mutable features would otherwise be challenging \[14\].

6. Dynamic Semantics

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

Semantics Objects. $\lambda_{te}$’s configuration consists of a store $s$, an expression $e$ 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 $s$ records a value, as well as the region $r$ and type $T$ of the reference. The trace records the runtime accesses to regions along the evaluation, with init$(r)$, rd$(r)$, and wr$(r)$, denoting the initialization, read, and write to $r$, respectively. Traces only serve a role in the soundness proofs, and thus are unnecessary in a $\lambda_{te}$ implementation. More specifically, we will show that the trace is the “realized effects” of the effects computed by $\lambda_{te}$.

The small-step semantics is defined as transition $s; e; f \rightarrow s'; e'; f'$. Given a store $s$ and a trace $f$, the evaluation of an expression $e$ results in another expression $e'$, a (possibly updated) store $s'$, and a trace $f'$. The notation $[x \backslash v]e$ substitutes $x$ with $v$ in expression $e$. 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 $\lambda_{te}$’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 $s; \Delta; \Gamma \vdash e : T, \sigma \sim \sigma'$, 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, $e$ in query $\ e$ 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., $\lambda_{te}$ is not an a posteriori effect monitoring system.

Applying full-fledged dynamic typing could be expensive. In §4, 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 $\Gamma$. 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 $\langle e, T, \sigma, \sigma' \rangle$ and evaluates the passenger expression $e$. To show the validity of the effect query in first-class effects, we will prove in Theorem 7.3 that if $e$ is evaluated, its effects will fall within the lower $\sigma$ and upper bound $\sigma'$ effects. When the evaluation terminates, it reduces to a value of type $T$, specified in the closure.

Effect-Guided Programming via Predicated Effect Inspection. The (Eфф) rule analyzes the type-and-effect of the closure. It searches the first matching target types and returns the corresponding branch expression $e_i$. 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 $e; e'$ or let $x = e'$ in $e; x$. This treatment lets the result of $e'$ 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 $\lambda_{te}$. The full proofs could be found in our report \[15\]. 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 $e'$, written $e' \triangleright \langle s, e, f \rangle$. 

\[\]
\[ \text{Definitions:} \]

\[ \begin{align*}
\text{trc} &::= \langle e, T, \sigma, \sigma' \rangle \\
\text{s} &::= \{ l \mapsto (e, v) \} \\
\text{f} &::= \pi(\rho) \\
\text{v} &::= b \mid \lambda x : T.e \mid l \mid \text{trc} \\
\text{E} &::= \text{let } e = \text{ref } \rho. \text{T } | e | \text{realize } E \\
\text{Evaluation relation:} \quad &\text{if } E \text{ then } e \text{ else } e' | \text{effcase } x = \text{E : T where } P \Rightarrow \circ \end{align*} \]

effect closure
store
trace
value
evaluation context

\[ \text{Dynamic Typing: } s; \Gamma \vdash e : T, \sigma \sim \sigma' \]

\[ \text{(ST-LOC) } \frac{\{ l \mapsto (e, v) \} \in s \quad \text{(ST-TYPE)} }{ s; \Gamma \vdash e : T, \sigma \sim \sigma' } \]

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

\[ \text{Evaluation relation: } s; e; f \rightarrow s'; e'; f' \]

\[ \begin{align*}
\text{(ext)} & \quad s; \text{E}[e]; f \rightarrow s'; \text{E}'[e']; f, f' \quad \text{if } s; e \rightsquigarrow s'; e'; f' \\
\text{(app)} & \quad s; \lambda x : T.e \rightsquigarrow s; [x]v[e]; \emptyset \\
\text{(let)} & \quad s; \text{let } x = v \text{ in } s; [x[v]; e]; \emptyset \\
\text{(if)} & \quad s; \text{if } b \text{ then } e_0 \text{ else } e_1 \rightsquigarrow s; e; \emptyset \\
\text{(get)} & \quad s; l \cdot v \rightsquigarrow s; [l \mapsto (e, v)]v; \emptyset \\
\text{(set)} & \quad s; \text{ref } l \cdot v \rightsquigarrow s; [l \mapsto (e, v)]l; \emptyset \\
\text{(ref)} & \quad s; \text{ref } e \cdot v \rightsquigarrow s; [l \mapsto (e, v)]l; \emptyset \\
\text{(par)} & \quad s; e; [e'] \rightsquigarrow s; e_0; \emptyset \\
\text{(qry)} & \quad s; \text{query } e \rightsquigarrow s; \langle e, T, \sigma, \sigma' \rangle; \emptyset \\
\text{(real)} & \quad s; \text{realize } \langle e, T, \sigma, \sigma' \rangle \rightsquigarrow s; e; \emptyset \\
\text{(effc)} & \quad s; \text{effcase } x = \langle e, T, \sigma, \sigma' \rangle; \emptyset \text{ where } P \Rightarrow e \rightsquigarrow s; \emptyset \\
\text{(rfrnt)} & \quad s; \{ T, \sigma_0 \sim \sigma_1 \} P \Rightarrow e \rightsquigarrow s; e; \emptyset \\
\end{align*} \]

**Figure 12. \( \lambda_{te} \) Operational Semantics.**

\[ \text{iff } \vdash e' : T, \sigma \sim \sigma', \emptyset; e'; \emptyset \rightarrow^* s; \text{E}[e]; f. \quad \text{When } e' \text{ is irrelevant, we shorten it as } \vdash^* s, e, f. \]

**THEOREM 7.2 (Type Soundness).** Given an expression \( e \), \( \vdash e : T, \sigma \sim \sigma' \), 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 \( f \), and an expression \( e = \text{E}[\text{query } e'] \). If \( s; e; f \rightarrow s; \text{E}[(e', T, \sigma, \sigma')]; f \rightarrow^* s'; \text{E}'[\text{realize } \langle e', T, \sigma, \sigma' \rangle]; f' \text{ then } s; \emptyset; \emptyset \vdash e' : T, \sigma \sim \sigma', \text{ and } s'; \emptyset; \emptyset \vdash e' : T, \sigma \sim \sigma'. \)

To prove trace consistency, we define \( \text{trace for expression} \).

**DEFINITION 7.4 (Trace from Effect Closure).** We say \( f \) is a trace for expression \( e \) under store \( s \), written \( f \prec \ll e, s, \gg \), \( \text{iff } s; e; f' \rightarrow^* s'; v; f' \).

We now define the soundness over traces:

**DEFINITION 7.5 (Trace Consistency).** We say \( e \) is trace-consistent if for any \( \ll s, T, e, f' \gg, \exists = \{ T, \sigma \sim \sigma' | P \}, \text{ and } f \prec \ll e, s, \gg, \text{ then } [\sigma[f]\sigma'[f]]P \text{ holds.} \)

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

8. **More Examples**

| application | predicate(s): | \\|---------|---|
|------------|-------------|
| priority scheduler | \( \ll \) | \\| consistent software update | \( \ll \) | \\| information security | \( \ll \) | \\| algorithm speculation | \( \ll \) | \\| atomicity | \( \ll \) | \\| testing | \( \ll \) | 

**Figure 13. Polaries for Client Predicates.**

In this section, we motivate \( \lambda_{te} \) 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 \( \lambda_{te} \) 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
run transaction update

let run1 = λprologue, epilogue, update.

case epilogue of

| h: t | effcase prologue, epilogue, update:
| EC(xl ~ xu), EC(yl ~ yu), EC(zi ~ zy) where xu # xu V zu # yu
| => (prologue # update V epilogue # update) realize update; realize epilogue
| default => h; run1 (query {realize prologue}); h) t update in

let run = λtransaction, update.
run1 (query 0) transaction update

let data = ref 0 in
let fun0 = ref λx. x := !x + 1 in
let fun1 = ref λx. 1 in
let transaction = query [fun0 data, fun1 data, !data] in
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 \(P\) 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 \(\sigma_i\) of the patch do not conflict (\(#\)) with the effects \(\sigma_p\) of code of the transaction before the update, noted as prologue, or effects \(\sigma_e\) 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 17. 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, \(w_{x_1}\), \(w_{x_2}\), the effects of the three blocks of the transaction are below, and √ and X represent the effects of the block conflict and do not conflict with the swapping update, respectively:

\[
\begin{align*}
!\text{fun0 data} & | \text{rd}_{i}, \text{rd}_{a} | \times \\
!\text{fun1 data} & | \text{rd}_{i}, \text{rd}_{a} | X \\
!\text{data} & | \text{rd}_{a} | √
\end{align*}
\]

The update is problematic after the first block, because \(\sigma_i\) conflicts with both \(\sigma_p\) and \(\sigma_e\), while it is okay after the second block because \(\sigma_i\) only conflicts with \(\sigma_p\), but not \(\sigma_e\).

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 λte’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
---------------------------------------------------------------------
let pw = ref 12 in
let close = λxEffcase x:

| EC(1~u) where wr, r<:1 => wr, r<:x | realization x
| default => pw := -9 in

---------------------------------------------------------------------
Third party libraries
---------------------------------------------------------------------
close (query pw := -9); // Safe Library
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] [55]. We use
the zeroing strategy [55]. At runtime, we programmatically
analyze the effects of the library and execute it only if it de-
strloys (overwrites) the password in the must-effect wr, r<:x
y, to avoid the recovery of the original password.

Double-Bounded Effects The must-may effect distinction
in λte is crucial for program correctness. In traditional
type-and-effect systems [37], effects are conservative ap-
proximations 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 4 is predicated by
the may-effect, i.e., wr, r<&:u. The may-effect view would
allow the case to be selected as long as u may write to re-
gion 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 pro-
grammers use the must-effects on line 3 to ensure that the
pw must be overridden. We explain how λte 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(1~u) where
wr, r<&:1 ∧ rd, r<&:u (u does not read x), 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 15.

<table>
<thead>
<tr>
<th>how</th>
<th>why</th>
</tr>
</thead>
<tbody>
<tr>
<td>zeroing</td>
<td>destroy/overwrite sensitive data</td>
</tr>
<tr>
<td>confidentiality</td>
<td>can not read sensitive data</td>
</tr>
<tr>
<td>integrity</td>
<td>can not write sensitive data</td>
</tr>
<tr>
<td>multiple accesses</td>
<td>can access a subset of data, but not all negative authorization</td>
</tr>
<tr>
<td>weak authorization</td>
<td>can only read non-sensitive data</td>
</tr>
<tr>
<td>overrideable policy</td>
<td></td>
</tr>
</tbody>
</table>

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

Summary The flexibility of λte is on par with other security
systems and λte shares the philosophy of improving the
flexibility of meta-level designs “hidden” behind effect sys-
tem [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 λte, programmers can flex-
ibly 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 pro-
graming applications, including efficient algorithmic spec-
culation, 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 influ-
enced 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 an-
notate an arbitrary expression as effect and upon the evalua-
tion of that expression, control is transferred to a matching
catch-like handling expression as a first-class value. Sim-
ilar 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
customized effect domains. A program element in their system may carry unknown effects, which may become gradually known at runtime. Long et al. [55] develop intentional 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, Nielsen 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].

### 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


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 expression 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 to an annotated expression. The two expressions are identical, except that the query in the “annotated expression” now takes the form of query

References:


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


Abstract Syntax in Optimized $\lambda_{\text{te}}$:

\[
\begin{align*}
\text{o} & ::= \text{b} \mid \lambda x : T.o \mid x \mid o \mid \text{let } x = o \text{ in } o | o \mid o \mid \text{ref } \rho T o \mid ! o | o := o \mid \text{realize } o \\
\text{query} (\overline{x} : T \triangleright T, \sigma \sim \sigma') & \mid \text{if } o \text{ then } o \text{ else } o \mid \text{effcase } x = o : T \text{ where } P \Rightarrow o
\end{align*}
\]

\[
\begin{align*}
\text{o} & ::= (o, T, \sigma, \sigma') \\
\text{os} & ::= \Gamma \rightarrow (\sigma, T) \text{ ov} \\
\text{ov} & ::= b \mid \lambda x : T.o \mid l \mid o \\
\text{O} & ::= - | O o | (O o) | \text{let } x = O \text{ in } o \mid \text{ref } \rho T O \mid ! O | O := o | ov := o \\
\text{if } O \text{ then } o \text{ else } o \mid \text{realize } O \mid \text{effcase } x = o : T \text{ where } P \Rightarrow o
\end{align*}
\]

**Gen function**

\[
\begin{align*}
\text{Gen(Bool)} & = \text{Bool} \\
\text{Gen(Ref, } T) & = \text{Ref}_{T} \\
\text{Gen(T} \stackrel{\sigma \sim \sigma}{\rightarrow} T') & = \text{Gen(T)} \stackrel{\sim \varsigma'}{\rightarrow} \text{Gen(T')} \\
\text{Gen(EC(T, } \sigma \sim \sigma')) & = \text{EC}(\text{Gen(T)}, \varsigma \sim \varsigma')
\end{align*}
\]

where $\varsigma, \varsigma'$ fresh

Transformation:

\[
\begin{align*}
\text{query } e & \Rightarrow o \\
\text{effcase } x & = e : T \text{ where } P \Rightarrow e \Rightarrow o \\
\text{realize } e & \Rightarrow o \\
\lambda x : T.e & \Rightarrow \lambda x : T.o \\
\text{let } x & = e \text{ in } e' \Rightarrow o \\
\text{if } e \text{ then } e_{0} \text{ else } e_{1} & \Rightarrow o
\end{align*}
\]

Evaluation relation:

\[
\begin{align*}
\text{os; } o & \rightarrow_{O} \text{os'; } o' \Rightarrow f' \\
\text{os; } O[o] ; f & \rightarrow_{O} \text{os'; } O[o'] ; f' \\
\text{(Octx)}
\end{align*}
\]

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

**Figure 19.** Optimized $\lambda_{te}$. 

\[
\overline{x} : T \triangleright T, \sigma \sim \sigma', \text{ o}
\]

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

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 $e$ to annotated expression $o$ under $\Delta$ and $\Gamma$, denoted as $e \Rightarrow_{\Delta, \Gamma} o$, 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 $\rightarrow_{\Delta, \Gamma}^{\bullet}$ represents the reflexive and transitive closure of $\rightarrow_{O}$. Upon the evaluation of the annotated query expression, the types associated with the free
B. More Applicability

B.1 Latent Effect Analysis

With $\lambda_{\text{ef}}$, 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 $\text{bf}0$, $\text{bf}1$, and $\text{bf}2$ in different regions $r$, $u$, and $w$ respectively. The function $s$ accepts a function $\text{fn}$ 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 (line 7), $O(n^2)$ Complexity. On the Right, Latent Effects Are Checked (line 16), $O(n)$ Complexity.

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). 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 $\text{os}; \Delta; \Gamma \vdash_{\text{ef}} o : \tau, \sigma \sim \sigma'$, defined as $s; \Delta; \Gamma \vdash e : \tau, \sigma \sim \sigma'$ where $e \sim_{\text{ef}} o$, substituted types of the free variables with the types of the values.

B.2 Algorithmic Speculation

We highlight the use of the non-monotone operator $\Rightarrow$ (line 4), in Figure 21, which is not supported or can be hard to reason about in previous system [35]. The operator $\Rightarrow$ 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 $\lambda_{\text{ef}}$, overhead can be further reduced.

If the to-be-speculated computation $x$ is pure and only writes the final result to the returning $\text{buf}$, then the copying overhead can be avoided. The programmer analyzes its effects (line 4), and if both its must- and may-effects are $\text{wr}_r$, i.e., $\text{EC}(\text{wr}_r \sim \text{wr}_r), x$ 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 $\text{buggy}$, if the value of $\text{buf}$ 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 [35].

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 [13][56]. Under the preemptive semantics, a $\text{yield}$ 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 = \x. for y : !threads
2. | effcase y:
3. | EC(h \sim z) where wr, <<: h \Rightarrow (wr, <<: y) realize y
4. let yield = \x. for y : !threads
5. | effcase y:
6. | EC(h \sim z) where wr, /<<: z \Rightarrow (wr, /<<: y) realize y

Application code

7. let buf = ref 0 in
8. let buggy = \x. if !buf not 0
9. then yield buf; /* break point */
10. 
11. let threads = ref (cons (query buggy 0) (query interfere 0)) in
12. buggy \| interfere

Yield in Atomicity

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].

\[ \lambda x_1, x_2, x_3. \]

1. effcase x1, x2, x3:
2. | EC(y4 \sim y0), EC(y5 \sim y1), EC(y6 \sim y2) where y0 \# y1 \land y0 \# y2 \land y1 \# y2 \Rightarrow realize x1 || realize x2 || realize x3
3. |
4. |
5. |
6. |
7. |
8. |
9. |
10. |

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

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 \( \lambda_{fe} \), programmers are endowed the power of implementing any of the schedulings and combining them in any order.

With \( \lambda_{fe} \), programmers can implement don’t-care non-determinism scheduling as in Figure [23].