Intensional Effect Polymorphism*

Yuheng Long[1], Yu David Liu[2], and Hridesh Rajan[3]

1,3 Iowa State University
226 Atanasoff Hall, Ames, IA 50011, USA
{csgzlong,hridesh}@iastate.edu
2 SUNY Binghamton
Binghamton NY 13902, USA
davidL@cs.binghamton.edu

Abstract

Type-and-effect systems are a powerful tool for program construction and verification. We describe *intensional effect polymorphism*, a new foundation for effect systems that integrates static and dynamic effect checking. Our system allows the effect of polymorphic code to be intensionally inspected through a lightweight notion of dynamic typing. When coupled with parametric polymorphism, the powerful system utilizes runtime information to enable precise effect reasoning, while at the same time retains strong type safety guarantees. We build our ideas on top of an imperative core calculus with regions. The technical innovations of our design include a *relational* notion of effect checking, the use of *bounded existential types* to capture the subtle interactions between static typing and dynamic typing, and a *differential alignment* strategy to achieve efficiency in dynamic typing. We demonstrate the applications of intensional effect polymorphism in concurrent programming, security, graphical user interface access, and memoization.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.2 Semantics of Programming Languages

Keywords and phrases intensional effect polymorphism, type system, hybrid typing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.346

1 Introduction

In a type-and-effect system [27, 36], the type information of expression e encodes and approximates the computational effects σ of e, such as how memory locations are accessed in e. Type-and-effect systems – or effect systems for short in this paper – have broad applications (*e.g.*, [2, 29, 24, 6]). Improving their expressiveness and precision through static approaches is a thoroughly explored topic, where many classic language design (*e.g.*, [22, 15, 35, 4, 26]) and program analysis (*e.g.*, [33, 3]) techniques may be useful.

Purely static effect systems are a worthy direction, but looking forward, we believe that a complementary *foundation* is also warranted, where the default is a system that can fully account for and exploit runtime information, aided by static approaches for optimization. Our belief is shaped by two insights. First, emerging software systems increasingly rely on dynamic language features: reflection, dynamic linking/loading, native code interface, flexible meta programming in script languages, to name a few. Second, traditional hurdles

* This work was partially supported by the US National Science Foundation (NSF) under grants 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.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 346–370

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

defying precise static reasoning – such as expression ordering, branching, recursion, and object dynamic dispatch – are often amplified in the context of effect reasoning.

In this paper, we develop *intensional effect polymorphism*, a system that integrates static and dynamic effect reasoning. The system relies on dynamic typing to compensate for the conservativeness of traditional static approaches and account for emerging dynamic features, while at the same time harvesting the power of static typing to vouchsafe for programs whose type safety is fundamentally dependent on runtime decision making. Consider the following example:

▶ **Example 1** (Conservativeness of Static Typing for Race-Free Parallelism)**.** Imagine we would like to design a type system to guarantee race freedom of parallel programs. Let expression e||e′ denote running e and e′ in parallel, whose typing rule requires that e and e′ have memory access effects over disjoint regions. Further, let `r1` and `r2` be disjoint regions. The following program is race-free, even though a purely static effect system is likely to reject it:

```
1                    (λx.λy. (x := 1) || !y)
2                            (if 1 > 0 then ref_r1 0 else ref_r2 0)
3                            (if 0 > 1 then ref_r1 0 else ref_r2 0)
```

Observe that parametric polymorphism is not helpful here: `x` and `y` can certainly be typed as region-polymorphic, but the program remains untypable. The root cause of this problem is that race freedom only depends on the *runtime* behaviors of `(x:=1) || !y`, which only depends on what `x` and `y` are *at runtime*.

Inspired by Harper and Morrisett [19], we propose an effect system where polymorphic code may intensionally inspect effects at runtime. Specifically, expression **assuming** e $\mathbb{R}$ e′ **do** e_1 **else** e_2 inspects whether the runtime (effect) type of e and that of e′ satisfy binary relation $\mathbb{R}$, and evaluates e_1 if so, or e_2 otherwise. Our core calculus leaves predicate $\mathbb{R}$ abstract, which under different instantiations can support a family of concrete type-and-effect language systems. To illustrate the example of race freedom, let us consider $\mathbb{R}$ being implemented as region disjointness relation #. The previous example can be written in our calculus as follows.

▶ **Example 2** (Intensional Effect Polymorphism for Race-Free Parallelism)**.** The following program type checks, with the static system and the dynamic system interacting in interesting ways:

```
1                  (λx.λy. assuming (x := 0) # !y do (x := 1) || y)
2                          (if 1 > 0 then ref_r1 0 else ref_r2 0)
3                          (if 0 > 1 then ref_r1 0 else ref_r2 0)
```

Static typing can guarantee that the lambda abstraction in the first line is well-typed regardless of how it is applied, good news for modularity. Dynamic typing provides precise typing for expression `(x := 0)` and expression `!y` – exploiting the runtime type information of `x` and `y` – allowing for a more precise disjointness check. Observe that in our calculus, the subexpression participating in the **assuming** check – `x := 0` here – does not have to be syntactically isomorphic to the expression in the **do** subexpression, `x := 1`. For simplicity, we omit the **else** expression in this example.

Technical Innovations

On the highest level, our system shares the philosophy with a number of type system designs hybridizing static checking and dynamic checking (*e.g.*, [14, 34, 18]), and some in the contexts of effect reasoning [5, 20]. To the best of our knowledge however, this is the first time intensional type analysis is applied to effect reasoning. This combination is powerful,

because not only effect reasoning can rely on runtime type information, but also parametric polymorphism is fully retained. For example, observe that in the example above, the types for `x` and `y` are parametric, not just "unknowns" or "dynamic". Let us look at another example:

▶ **Example 3** (Parametric Polymorphism Preservation)**.** For the following program, the parallel execution in the second line is statically guaranteed to be type-safe in our system. Programs written with intensional effect polymorphism do not have runtime type errors.

```
1          let s = λx.λy. assuming (x := 0) # !y do (x := 1) || !y in
2                    (s ref_r1 0 ref_r2 0) || (s ref_r3 0 ref_r4 0)
```

In addition, intensional effect polymorphism goes beyond a mechanical adaptation of Harper-Morrisett, with several technical innovations we now summarize. The most remarkable difference is that the intensionality of our type system is enabled through *dynamic typing*. At runtime, the evaluation of expression **assuming** e $\mathbb{R}$ e′ **do** e_1 leads to the dynamic typing of e and e′. In contrast, the classic intensional type analysis performs a `typecase`-like inspection on the runtime instantiation of the polymorphic type. Our strategy is more general, in that it not only subsumes the former – indeed, a type derivation conceptually constructed at runtime must have leaf nodes as instances of value typing – but also allows (the effect of) arbitrary expressions to be inspected at runtime. We believe this design is particularly relevant for effect reasoning, because it has less to do with the effect of polymorphically-typed variables, and more with *where* these variables appear in the program.

Second, we design the runtime type inspection through a *relational* check. In the **assuming** expression, our system dynamically checks whether $\mathbb{R}$ holds, instead of computing what the effect of e or e′ is. The relational design does not require programmers to explicitly provide an "effect specification/pattern" of the runtime type – a task potentially daunting as it may either involve enumerating region names, or expressing conditional specifications such as "a region that some other expression does not touch." Many safety properties reasoned about by effect systems are relational in nature, such as thread interference.

Third, the subtle interaction between static typing and dynamic typing poses a unique challenge on type soundness in the presence of effect subsumption. We elaborate on this issue in Section 4.4. We introduce a notion of *bounded existential types* to differentiate but relate the types assumed by the static system and those by the dynamic system.

Finally, a full-fledged construction of type derivations at runtime for dynamic typing would incur significant overhead. We design a novel optimization to allow for efficient runtime effect computation, eliminating the need for dynamic derivation construction while producing the same result. The key insight is that we could align the static type derivation and the (would-be-constructed) dynamic type derivation, and compute the effects of the latter simply by substituting the difference of the two, a strategy we call *differential alignment*. We will detail this design in Section 5.

We formalize intensional effect polymorphism in λ_{ie}, an imperative call-by-value λ-calculus with regions. In summary, this paper makes the following contributions:

- It describes a hybrid type system for effect reasoning centering on intensional polymorphism.
- It develops a sound type system and operational semantics where relational effect inspection is made abstract.
- It illuminates the subtleties resulting from the difference between static effect reasoning and dynamic effect reasoning, and proposes bounded existential types to preserve soundness, and differential alignment to promote efficiency.
- It demonstrates the impact of our design by extending the core calculus to the applications of safe parallelism support, security, UI access, and memoization.

```
1  class Pair {
2    int ft = 1, sd = 2;

4    int applyTwice(Op f) {
5      assuming ft = f.op(0)   #   sd = f.op(5)
6      do ft = f.op(f.op(ft)) || sd = f.op(f.op(sd));
7    else ft = f.op(f.op(ft)) ;   sd = f.op(f.op(sd));
8    }
9  }

11 Pair pr = new Pair();
12 Op o = (Op) newInstance(readFile("filePath"));
13 pr.applyTwice(o);
```

```
14 interface Op { int op(int i); }

16 class Pref implements Op {
17   int sum = 0;?
18   // effect:  write sum
19   int op(int i) { sum += i; }
20 }

22 class Hash implements Op {
23   // effect:  pure, no effect
24   int op(int i) { hash(i); }
25 }
```

Figure 1 An application of intensional effects in enforcing safe parallelism.

2 Motivating Examples

In this section, we demonstrate the applicability of intensional effects in reasoning about safe parallelism, information security, consistent UI access, and program optimization. In each of these applications, the refined notion of type safety is fundamentally dependent on runtime decision making, *i.e.*, whether the relation $\mathbb{R}$ is satisfied. We instantiate the effect relation operator $\mathbb{R}$ with different concrete relations between effects of expressions.

As in previous work [17, 8], we optionally extend standard Java-like syntax with *region declarations* when the client language deems them necessary. In that case, a variable declaration may contain both type and region annotations, *e.g.*, `JLabel j` **in** `ui` declares a variable `j` in region `ui`. For client languages where regions are not explicitly annotated, different abstract locations (such as different fields of an object) are treated as separate regions.

2.1 Safe Parallelism

We demonstrate the application of intensional effects in supporting safe parallelism, where safety in this context refers to the conventional notion of thread non-interference (race freedom) [27]. Concretely, Figure 1 is a simplified example of "operation-agnostic" data parallelism, where the programmer's intention is to apply some statically unknown operation (encapsulated in an `Op` object) – here implemented through reflection – to a data set, here simplified as a pair of data `ft` and `sd`. The programmer wishes to "best effort" leverage parallelism to process `ft` and `sd` in parallel, without sacrificing thread non-interference. The tricky problem of this notion of safety is it depends on what `Op` object is. For instance, parallel processing of the pair with the `Hash` object is safe, but not when the operation at concern is the prefix sum operator [7], encapsulated as `Pref`.

Static reasoning about the correctness of the parallel composition could be challenging in this example, because the `Op` object remains unknown until `applyTwice` is invoked at runtime.

The **assuming** expression (line 5) helps the program retain strong type safety guarantees for parallel composition (line 6), while utilizing the runtime information to enable precise reasoning. At runtime, the **assuming** expression intensionally inspects the effects of the expressions `ft = f.op(0)` and `sd = f.op(5)`. If they satisfy the binary relation #, parallelism will be enabled. If `f` points to a `Hash` object, the # relation will be true and the program enjoys safe concurrency (line 6). On the other hand, if `f` points to a `Pref` object, the program will be run sequentially, desirable for race freedom safety.

```
1  class Page {
2    String searchBox = "";
3    String url = "wsj.com/search?";
4    String location = "";

6    String load_adv(ThirdParty adv) {
7      assuming url ◊ adv.show(this)
8      do exec url adv.show(this);
9      else "no advertisement";
10   }

12   int search(ThirdParty adv) {
13     load_adv(adv);
14     location = url + searchBox;
15   }
16 }
```

```
17 interface ThirdParty { String show(Page p); }

19 class Good implements ThirdParty {
20   String show(Page p) { "404"; }
21 }

23 class Evil implements ThirdParty {
24   String show(Page p) {
25     p.url = "evil.com";
26   }
27 }

29 ThirdParty adv = (ThirdParty)
30     newInstance(readFile("filePath"));
31 new Page().render(adv);
```

Figure 2 An application of intensional effects in preventing security vulnerabilities.

2.2 Information Security

As another application of intensional effects, consider its usage in preventing security vulnerabilities. Figure 2 presents an adapted (`wsj.com`) example of real-world security vulnerabilities [11]. The `Page` class allows users to search information within the site. Once the `search` is called, the page will redirect to a web page corresponding to the `url` and `searchBox` strings (the redirection is represented as changing the `location` variable for simplicity). The page, when created, inserts a third party advertisement (line 8).

The third party code can be malicious, *e.g.*, it can modify the search `url` and redirects the search to a malicious site, from which the whole system could be compromised, *e.g.*, the `Evil` third-party code. Ensuring the key security properties becomes challenging with the dynamic features because the third party code is only available at runtime, loaded using reflection. The expression **exec** e_1 e_2 (line 7) encodes a *check-then-act* programming pattern. It executes e_2 only if it does not read nor write any object accessible by e_1 and otherwise it gets stuck.

With intensional effects, users can intensionally inspect a third party code e whenever e is dynamically loaded. The intensional inspection, accompanied with a relational policy check, ensures that e does not access any sensitive data (the `url`), specified using the relation $\Diamond$. It also ensures that the **exec** expression does not get stuck.

2.3 Consistent Graphical User Interface (GUI) Access

We show how intensional effects can be used to reason about the correctness of a GUI usage pattern, common in Subclipse, JDK, Eclipse and JFace [16]. Typically, GUI has a single *UI thread* handling events in the "event loop". This UI thread often spawns separate *background threads* to handle time-consuming operations. Many frameworks enforce a single-threaded GUI policy: only the UI thread can access the GUI objects [16]. If this policy is violated, the whole application may abort or crash. Figure 3 shows a simplified example of a UI thread that pulls an event from the `eventloop` and handles it. In the application, all UI elements reside in the `ui` region (declared at line 14), *e.g.*, the field `j` at line 23.

The safety here refers to no UI access in any background thread. The tricky problem here is that the events arrive at runtime with different event handlers. Some handlers may access UI objects while the others do not. Therefore, the correctness of spawning a thread to handle a new event, depends heavily on what objects the corresponding event handler has. For instance, the handler containing a `NonUI` object can be executed in a background thread, while `UIAccess` should not. The expression **spawn** e_1 e_2, executes e_2 in a background thread

```
1  class UIThread {
2    JLabel global in ui = new JLabel();
3    void eventloop(Runnable closure) {
4      assuming global ∅ closure.run()
5        do spawn global closure.run();
6      else closure.run();
7    }
8  }

10 Runnable closure;
11 if (1 > 0) closure = new NonUI();
12 else closure = new UIAccess();
13 new UIThread().eventloop(closure);

14 region ui;

16 interface Runnable { String run(); }

18 class NonUI implements Runnable {
19   String run() { "does nothing"; }
20 }

22 class UIAccess implements Runnable {
23   JLabel j in ui = new JLabel();
24   String run() { j.val = "UI"; }
25 }
```

Figure 3 An application of intensional effects in disciplining UI access.

```
1  class Mem {
2    Integer input = new Integer();

4    int comp(Mutate m, Integer x) {
5      int cache = heavy(input);
6      assuming m.mutate(x) ♮ heavy(input)
7      do lookup m.mutate(x) heavy(input); cache
8      else m.mutate(x); heavy(input);
9    }

11   int heavy(Integer i) { /* ... */}
12 }

13 class Integer { int i = 0; }

15 class Mutate {
16   int mutate(Integer input) {
17     input.i = 101;
18   }
19 }

21 Memo mm = new Mem();
22 Mutate mu = new Mutate();
23 if (1 > 0) mm.comp(mu, mm.input);
24 else mm.comp(mu, new Integer());
```

Figure 4 An application of intensional effects in providing effective memoization.

only if it does not allocate, read or write any object in the region accessed by e_1. Otherwise it gets stuck.

The **assuming** expression, used by the UI thread, statically guarantees strong type safety for the **spawn** expression, so it does not get stuck. It also utilizes precise runtime information to distinguish handlers with no UI accesses from other handlers. If a handler satisfies relation ∅, it can be safely executed by a background thread. The relation ∅ is satisfied if the RHS expression does not allocate, read/write any region denoted by the LHS expression.

2.4 Program Optimization – Memoization

As another application, we utilize intensional effects to implement a proof-of-concept memoization technique in a sequential program. Memoization is an optimization technique where the results of expensive function calls are cached and these cached results are returned when the inputs and the environment of the function are the same.

Figure 4 presents a simplified application where repeated tasks, here the `heavy` method calls on line 5 and 8, are performed. These two tasks are separated by a small computation `mutate`, forming a *compute-mutate* pattern [9]. We leave the body of the method `heavy` intentionally unspecified, which could represent a set of computationally expensive operations. It could, *e.g.*, generate the power set `ps` of a set of `input` elements and return the size of `ps` or do the Bogosort.

The second `heavy` task needs not be recomputed in full if the `mutate` invocation does not modify the input nor the environment of `heavy`. If so, the cached result of the first call can be reused and the repeated computation can be avoided. The expression **lookup** e_1 e_2 executes the expressions e_1 only if e_1 does not write to objects in the regions read by e_2. Otherwise it gets stuck.

Ensuring that the **lookup** expression does not get stuck is challenging. This is because the validity of the cached result depends on the runtime value of both the mutation `m` and

$$
\begin{array}{lcll}
v & ::= & \mathrm{b} \mid \lambda \mathrm{x} : \mathrm{T}.\mathrm{e} & \textit{value} \\
\mathrm{e} & ::= & v \mid \mathrm{x} \mid \mathrm{e}\ \mathrm{e} \mid \textbf{let}\ \mathrm{x} = \mathrm{e}\ \textbf{in}\ \mathrm{e} \mid \textbf{if}\ \mathrm{e}\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e} & \textit{expression} \\
 & & \mid \textbf{ref}\ \rho\ \mathrm{T}\ \mathrm{e} \mid\ !\mathrm{e} \mid \mathrm{e}\ \texttt{:=}\ \mathrm{e} & \\
 & & \mid \textbf{assuming}\ \mathrm{e}\ \mathbb{R}\ \mathrm{e}\ \textbf{do}\ \mathrm{e}\ \textbf{else}\ \mathrm{e} \mid \mathbb{SAFE}\ \mathrm{e}\ \mathrm{e} & \\
\mathrm{T} & ::= & \alpha \mid \textbf{Bool} \mid \mathrm{T} \xrightarrow{\sigma} \mathrm{T}' \mid \textbf{Ref}_{\rho}\ \mathrm{T} & \textit{type} \\
\rho & ::= & \overline{\zeta} & \textit{region} \\
\zeta & ::= & \mathrm{r} \mid \gamma & \textit{region element} \\
\sigma & ::= & \overline{\omega} & \textit{effect} \\
\omega & ::= & \varsigma \mid \mathit{acc}_{\rho}\mathrm{T} & \textit{effect element} \\
\mathit{acc} & ::= & \textbf{init} \mid \textbf{rd} \mid \textbf{wr} & \textit{access right}
\end{array}
$$

Figure 5 λ_{ie} Abstract Syntax. (Throughout the paper, notation $\overline{\bullet}$ represents a set of $\bullet$ elements, and notation $\vec{\bullet}$ represents a sequence of $\bullet$ elements.)

its input `x`. For example, if the parameter `x` is a new object as the one created at line 24, the cache is valid, while the one represented by `mm.input` (line 23) is not valid.

The **assuming** expression solves the problem: the safety of the **lookup** expression is statically guaranteed. At runtime, with precise dynamic information, the intensional binary $\natural$ relational check ensures that the write accesses of the LHS do not affect the RHS expression. If this relation is satisfied, the cache is valid and can be reused.

Other optimizations

Intensional effect polymorphism can be used for other similar optimizations, *e.g.*, record-and-reply style memoization, common sub-expression elimination, loop-invariant code motion and redundant load elimination. In all these applications, if the mutation, similar to the style of, for example, `m.mutate(x)`, is infrequent or does not modify a large portion of the heap, the cached results can avoid repeated expensive computations.

Summary

The essence of intensional effect polymorphism lies in the *interesting interplay between static typing and dynamic typing*. Static typing guarantees that the potentially unsafe expressions are only used under runtime "safe" contexts (*i.e.*, those that pass the relational effect inspection), in highly dynamic scenarios such as parallel composition, loading third party code, handling I/O events, and data reuse. Dynamic typing exploits program runtime type information to allow for more precise effect reasoning, in that "safe" contexts can be dynamically decided upon based on runtime type/effect information.

3 λ_{ie} Abstract Syntax

To highlight the foundational nature of intensional effect polymorphism, we build our ideas on top of an imperative region-based lambda calculus. The abstract syntax of λ_{ie} is defined in Figure 5. Expressions are standard for an imperative λ calculus, except the last two forms which we will soon elaborate. We do not model integers and unit values, even though our examples may freely use them. Since **if** e **then** e **else** e plays a non-trivial role in our examples, we choose to model it explicitly. As a result, boolean values $\mathrm{b} \in \{\mathit{true}, \mathit{false}\}$ are also explicitly modeled. Metavariable x represents variables.

Our core syntax is expressive enough to encode the core program logic of the examples in Section 2. However, it does not model objects for simplicity without the loss of generality. An extension with objects and classes is mostly standard [30] and is included in our technical report [25].

We introduced expression **assuming** e $\mathbb{R}$ e′ **do** e_0 **else** e_1, where from now on we call e and e′ the condition expressions, e_0 the **do** expression, and e_1 the **else** expression. At runtime, this expression retrieves the effects of e and e′ through dynamic typing, without evaluating e or e′. The timing of gaining this knowledge is important: the conditions will not be evaluated and the **do** expression is not evaluated yet. In other words, even though our system relies on runtime information, it is not an *a posteriori* effect monitoring system.

A General Framework

As illustrated in Section 2, effect reasoning has diverse applications. We aimed to design a *general* framework for effect reasoning, which can be concretized to different "client" languages. To achieve this goal, we choose to (1) leave the definition of the binary relation $\mathbb{R}$ abstract; (2) include an abstract $\mathbb{SAFE}$ e e′ expression, which is type-safe iff e $\mathbb{R}$ e′ holds. The $\mathbb{R}$ relation and the $\mathbb{SAFE}$ expression can be concretized to different "client" languages to capture different application domain goals. For example, when $\mathbb{R}$ is concretized to thread non-interference, one possible concretization of $\mathbb{SAFE}$ e e′ is parallel composition e||e′. The instantiations of $\mathbb{R}$ of the applications in Section 2 are shown in Figure 6. The only requirement for $\mathbb{R}$ is that it must be monotone [28, 8], *i.e.*, closed under effect subsetting. Concretely, it says that if σ $\mathbb{R}$ σ', $\sigma_0 \subseteq \sigma$ and $\sigma_1 \subseteq \sigma'$, then σ_0 $\mathbb{R}$ σ_1. All instantiations in Section 2 satisfy this requirement.

Types, Regions, and Effects

Programmer types are either primitive types, reference types $\mathbf{Ref}_\rho$ T for store values of type T in region ρ, or function types $T \xrightarrow{\sigma} T'$, from T to T′ with σ as the effect of the function body. Last but not least, as a framework with parametric polymorphism, types may be type variables α.

Our notion of regions is standard [36, 27], an abstract collection of memory locations. A region in our language can either be demarcated as a constant r, or parametrically as a region variable γ.

An effect is a set of effect elements, either an effect variable ς, or acc_ρT, representing an access *acc* to region ρ whose stored values are of type T. Access rights **init**, **rd**, **wr** represent allocation, read, and write, respectively.

As the grammar suggests, our framework is a flexible system where a type, a region, or an effect may all be parametrically polymorphic.

4 The Type System

This section describes the static semantics to our type-and-effect system. Overall, the type system associates each expression with effects, a goal shared by all effect systems. The highlight is how to construct a *precise* and *sound* effect system to support dynamic-typing-based intensionality. The precision of this type system is rooted at the $\mathbb{R}$ relation enforcement, at **assuming** time, based on effects computed by dynamic typing over runtime values and their types. Our static type system is designed so that any $\mathbb{SAFE}$ expression appearing in the **do** branch does not need to resort to runtime enforcement and the $\mathbb{R}$ relation is guaranteed

Safe Parallel Composition, Section 2.1

e $\mathbb{R}$ e′ $\stackrel{\text{def}}{=}$ e#e′ "Two effects do not interfere."

$\mathbb{SAFE}$ e e′ $\stackrel{\text{def}}{=}$ (e || e′) "Run the two expressions in parallel."

is defined as:

$$\emptyset \# \sigma \qquad \frac{\sigma \# \sigma'' \quad \sigma' \# \sigma''}{\sigma \cup \sigma' \# \sigma''} \qquad \frac{\sigma' \# \sigma}{\sigma \# \sigma'} \qquad \mathbf{rd}_\rho \mathrm{T} \# \mathbf{rd}_{\rho'} \mathrm{T}' \qquad \frac{\rho \neq \rho'}{\mathbf{rd}_\rho \mathrm{T} \# \mathbf{wr}_{\rho'} \mathrm{T}'} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}_\rho \mathrm{T} \# \mathbf{wr}_{\rho'} \mathrm{T}'}$$

Information Security, Section 2.2

e $\mathbb{R}$ e′ $\stackrel{\text{def}}{=}$ e$\diamondsuit$e′ "Expression e′ does not read/write regions accessible by e."

$\mathbb{SAFE}$ e e′ $\stackrel{\text{def}}{=}$ **exec** e e′ "Execute e′ if it does not read/write the regions by e."

$\diamondsuit$ is defined as:

$$\sigma \diamondsuit \emptyset \qquad \frac{\sigma \diamondsuit \sigma'' \quad \sigma' \diamondsuit \sigma''}{\sigma \cup \sigma' \diamondsuit \sigma''} \qquad \frac{\sigma'' \diamondsuit \sigma \quad \sigma'' \diamondsuit \sigma'}{\sigma'' \diamondsuit \sigma \cup \sigma'} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathrm{T} \diamondsuit \mathbf{rd}_{\rho'} \mathrm{T}'} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathrm{T} \diamondsuit \mathbf{wr}_{\rho'} \mathrm{T}'}$$

UI Access, Section 2.3

e $\mathbb{R}$ e′ $\stackrel{\text{def}}{=}$ e$\oslash$e′ "Expression e′ does not access regions accessible by e."

$\mathbb{SAFE}$ e e′ $\stackrel{\text{def}}{=}$ **spawn** e e′ "Execute e′ in another thread if it accesses no region by e."

$\oslash$ is defined as:

$$\sigma \oslash \emptyset \qquad \frac{\sigma'' \oslash \sigma \quad \sigma'' \oslash \sigma'}{\sigma'' \oslash \sigma \cup \sigma'} \qquad \frac{\sigma \oslash \sigma'' \quad \sigma' \oslash \sigma''}{\sigma \cup \sigma' \oslash \sigma''} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathrm{T} \oslash acc_{\rho'} \mathrm{T}'}$$

Memoization, Section 2.4

e $\mathbb{R}$ e′ $\stackrel{\text{def}}{=}$ e$\natural$e′ "RHS's read has no dependcy on the LHS's write"

$\mathbb{SAFE}$ e e′ $\stackrel{\text{def}}{=}$ **lookup** e e′ "Execute e if e writes no region read by e′."

$\natural$ is defined as:

$$\emptyset \natural \sigma \qquad \frac{\sigma \natural \sigma'' \quad \sigma' \natural \sigma''}{\sigma \cup \sigma' \natural \sigma''} \qquad \mathbf{rd}_\rho \mathrm{T} \natural \sigma \qquad \sigma \natural \mathbf{wr}_\rho \mathrm{T} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}_\rho \mathrm{T} \natural \mathbf{rd}_{\rho'} \mathrm{T}'}$$

Figure 6 Client Implementation of $\mathbb{R}$ and $\mathbb{SAFE}$ e e.

$$
\begin{array}{lcll}
\Gamma & ::= & \overrightarrow{\mathrm{x} \mapsto \tau} & \textit{type environment} \\
\tau & ::= & \forall \vec{g}.\exists\Sigma.\mathrm{T} & \textit{type scheme} \\
g & ::= & \alpha \mid \gamma \mid \varsigma & \textit{generic variable} \\
gs & ::= & \mathrm{T} \mid \rho \mid \sigma & \textit{generic structure} \\
\Phi & ::= & \overline{\Lambda} & \textit{relationship set} \\
\Sigma & ::= & \overline{g \preceq: gs} & \textit{subsumption set} \\
\Lambda & ::= & \sigma\ \mathbb{R}\ \sigma \mid \forall \vec{g}.\Sigma & \textit{relationship}
\end{array}
$$

Figure 7 λ_{ie} Type System Definitions.

to hold by the static type system. As we shall see, this leads to non-trivial challenges to soundness, as static typing and dynamic typing make related – yet different – assumptions on effects.

4.1 Definitions

Relevant structures of our type system are defined in Figure 7.

Type Environment and Type Scheme

Type environment Γ maps variables to type schemes, and we use notation $\Gamma(x)$ to refer to T where the rightmost occurrence of $\mathrm{x} : \mathrm{T}'$ for any T' in Γ is $\mathrm{x} : \mathrm{T}$.

A type scheme is similar to the standard notion where names may be bound through quantification [13]. Our type scheme, in the form of $\forall \vec{g}.\exists\Sigma.\mathrm{T}$, supports both universal quantification and existential quantification. Our use of universal quantification is mundane: the same is routinely used for parametric polymorphism systems. Observe that in our system, type variables, region variables, and effect variables may all be quantified, and we use a metavariable g for this general form, and call it a *generic variable*. Similarly, we use a unified variable gs to represent either a type, a region, or an effect, and call it a *generic structure* for convenience. Existential quantification is introduced to maintain soundness, a topic we will elaborate in a later subsection. For now, only observe that existentially quantified variables appear in the type scheme as a sequence of $g \preceq: gs$, each of which we call a *subsumption relationship*. Here we also informally say g is existentially quantified, with *bound* gs. When $\vec{g}$ is a sequence of 0 and Σ is empty, we also shorten the type scheme $\forall \vec{g}.\exists\Sigma.\mathrm{T}$ as T. Type schemes are alpha-equivalent.

Relationship Set

Another crucial structure to construct our type system is the *relationship set* Φ. On the high level, this structure captures the relationships between generic structures. Concretely, it is represented as a set whose element may either be an *abstract effect relationship* $\sigma\ \mathbb{R}\ \sigma'$ – denoting two effects σ and σ' conform to the $\mathbb{R}$ relation – or a *subsumption context relationship*. The latter is represented as $\forall \vec{g}.\Sigma$. Intuitively, a subsumption context relationship is a collection of subsumption relationships, except some of its generic variables may be universally quantified. Subsumption context relationships are alpha-equivalent.

As we shall see, the relationship set plays a pivotal role during type checking. At each step of derivation, this structure represents what one can assume about effects. For example, the interplay between **assuming** and $\mathbb{SAFE}$ is represented through whether the relationship set constructed through typing **assuming** can entail the relationship that makes the $\mathbb{SAFE}$ expression type-safe. Our relationship set may have a distinct structure, but effect system

Subtyping: $\Phi \vdash \mathrm{T} \preceq: \mathrm{T'}$

$$\text{(Type-refl)}\ \Phi \vdash \mathrm{T} \preceq: \mathrm{T} \qquad \text{(Type-tran)}\ \frac{\Phi \vdash \mathrm{T} \preceq: \mathrm{T}_0 \quad \Phi \vdash \mathrm{T}_0 \preceq: \mathrm{T}'}{\Phi \vdash \mathrm{T} \preceq: \mathrm{T}'} \qquad \text{(Type-ref)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho'}{\Phi \vdash \mathbf{Ref}_{\rho}\ \mathrm{T} \preceq: \mathbf{Ref}_{\rho'}\ \mathrm{T}}$$

$$\text{(Type-fun)}\ \frac{\Phi \vdash \mathrm{T}'_0 \preceq: \mathrm{T}_0 \qquad \Phi \vdash \mathrm{T}_1 \preceq: \mathrm{T}'_1 \qquad \Phi \vdash_{eff} \sigma \preceq: \sigma'}{\Phi \vdash \mathrm{T}_0 \xrightarrow{\sigma} \mathrm{T}_1 \preceq: \mathrm{T}'_0 \xrightarrow{\sigma'} \mathrm{T}'_1}$$

Effect Subsumption: $\Phi \vdash_{eff} \sigma \preceq: \sigma'$

$$\text{(Eff-refl)}\ \Phi \vdash_{eff} \sigma \preceq: \sigma \qquad \text{(Eff-tran)}\ \frac{\Phi \vdash_{eff} \sigma \preceq: \sigma_0 \qquad \Phi \vdash_{eff} \sigma_0 \preceq: \sigma'}{\Phi \vdash_{eff} \sigma \preceq: \sigma'} \qquad \text{(Eff-sub)}\ \frac{\sigma \subseteq \sigma'}{\Phi \vdash_{eff} \sigma \preceq: \sigma'} \qquad \text{(Eff-Cons)}\ \frac{\sigma \preceq: \sigma' \in \Phi}{\Phi \vdash_{eff} \sigma \preceq: \sigma'}$$

$$\text{(Eff-Acc)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho'}{\Phi \vdash_{eff} \{acc_{\rho}\mathrm{T}\} \preceq: \{acc_{\rho'}\mathrm{T}\}} \qquad \text{(Eff–Inst)}\ \frac{\forall \vec{g}.\Sigma \in \Phi \qquad \sigma \preceq: \sigma' \in \theta\Sigma \text{ for some } \theta \qquad dom(\theta) = \vec{g} \qquad ran(\theta) \cap ftv(\Sigma) = \emptyset}{\Phi \vdash_{eff} \sigma \preceq: \sigma'}$$

Region Subsumption: $\Phi \vdash_{reg} \rho \preceq: \rho'$

$$\text{(Reg-refl)}\ \Phi \vdash_{reg} \rho \preceq: \rho \qquad \text{(Reg-trans)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho_0 \qquad \Phi \vdash_{reg} \rho_0 \preceq: \rho'}{\Phi \vdash_{reg} \rho \preceq: \rho'} \qquad \text{(Reg-Sub)}\ \frac{\rho \subseteq \rho'}{\Phi \vdash_{reg} \rho \preceq: \rho'} \qquad \text{(Reg-Cons)}\ \frac{\rho \preceq: \rho' \in \Phi}{\Phi \vdash_{reg} \rho \preceq: \rho'}$$

$$\text{(Reg–Inst)}\ \frac{\forall \vec{g}.\Sigma \in \Phi \qquad \rho \preceq: \rho' \in \theta\Sigma \text{ for some } \theta \qquad dom(\theta) = \vec{g} \qquad ran(\theta) \cap ftv(\Sigma) = \emptyset}{\Phi \vdash_{reg} \rho \preceq: \rho'}$$

Relationship Entailment: $\Phi \vdash_{ar} \Lambda$

$$\text{(Rel–In)}\ \frac{\Lambda \in \Phi}{\Phi \vdash_{ar} \Lambda} \qquad \text{(Rel-Closed)}\ \frac{\Phi \vdash_{ar} \sigma\ \mathbb{R}\ \sigma' \qquad \Phi \vdash_{eff} \sigma_0 \preceq: \sigma \qquad \Phi \vdash_{eff} \sigma_1 \preceq: \sigma'}{\Phi \vdash_{ar} \sigma_0\ \mathbb{R}\ \sigma_1}$$

Figure 8 λ_{ie} Subsumption and Entailment.

designers should be able to find conceptual analogies in existing systems, such as privileges in Marino *et al.* [28].

Notations and Convenience Functions

We use (overloaded) function *ftv* to compute the set of free (*i.e.*, neither universally bound nor existentially bound) variables in T, ρ and σ. We use $fv(\mathrm{e})$ to compute the set of free variables in expression e. We use *dom* and *ran* to compute the domain and the range of a function. All definitions are standard. Substitution θ is a mapping function from type variables α to types T, region variables γ to regions ρ and effect variables ς to effects σ. The composition of substitutions, written $\theta\theta'$ is defined as $\theta\theta'(g) = \theta(\theta'(g))$.

We use comma for sequence concatenation. For example, $\Gamma, \mathrm{x} \mapsto \tau$ denotes appending sequence Γ with an additional binding from x to τ.

4.2 Subsumption and Entailment

Figure 8 defines subsumption relations for types, effects, and regions. All three forms of subsumption are reflexive and transitive. For function types, both return types and effects are covariant, whereas argument types are contra-variant. For **Ref** types, the regions are covariant, whereas the types for what the store holds must be invariant [37].

(Eff-Inst) and (Reg-Inst) capture the instantiation of universal variables in subsumption context relationship. After all, the latter is a collection of "parameterized" subsumption relationships which can be instantiated.

Finally, we define a simple relation $\Phi \vdash_{ar} \Lambda$ to denote that relationship set Φ can entail Λ. (Rel-In) says any relationship set may entail its element. (Rel-Closed) intuitively says that $\mathbb{R}$ is closed under taking subsetting. (Rel-Closed) is a manifestation of the monotonic requirement [28, 8] for $\mathbb{R}$, introduced in Section 3.

4.3 Typing Judgment Overview

Typing judgment in our system takes the form of $\Phi; \Gamma \vdash \mathrm{e} : \mathrm{T}, \sigma$, which consists of a type environment Γ, a relationship set Φ, an expression e, its type T and effect σ. When the relationship set and the type environment are empty, we further shorten the judgment as $\vdash \mathrm{e} : \mathrm{T}, \sigma$ for convenience. The judgment is defined in Figure 9, with auxiliary definitions related to universal and existential quantification deferred to Figure 10.

The rules (T-let) and (T-var) follow the familiar *let-polymorphism* (or Damas-Milner polymorphism [13]). Universal quantification is introduced at **let** boundaries, through the function $Gen(\Gamma, \sigma)(\mathrm{T})$. Its elimination is performed at (T-var), via $\preceq$. Both definitions are standard, and appear in Figure 10. The let-polymorphism in **let** x = e **in** e′ expression is sound because of the *Gen* function in the rule (T-let). The *Gen* function enforces the standard value restriction [36]. That is, if e is a value, its type could be generalized and thus be polymorphic, otherwise its type will be monomorphic.

(T-sub) describes subtyping, where both (monomorphic) type subsumption and effect subsumption may be applied. Rules (T-ref), (T-get) and (T-set) for store operations produce the effects of access rights **init**, **rd** and **wr**, respectively. All other rules other than (T-Assume) and (T-Safe) carry little surprise for an effect system.

4.4 Static Typing for Dynamic Intensional Analysis

To demonstrate how intensional effect analysis works, let us first consider an unsound but intuitive notion of **assuming** typing in (T-assume-unsound) (below).

$$\text{(T-assume-unsound)}\quad \frac{\Phi; \Gamma \vdash \mathrm{e} : \mathrm{T}, \sigma \qquad \Phi; \Gamma \vdash \mathrm{e}' : \mathrm{T}', \sigma' \qquad \Phi, \sigma\ \mathbb{R}\ \sigma'; \Gamma \vdash \mathrm{e}_0 : \mathrm{T}'', \sigma_0 \qquad \Phi; \Gamma \vdash \mathrm{e}_1 : \mathrm{T}'', \sigma_1}{\Phi; \Gamma \vdash \textbf{assuming}\ \mathrm{e}\ \mathbb{R}\ \mathrm{e}'\ \textbf{do}\ \mathrm{e}_0\ \textbf{else}\ \mathrm{e}_1 : \mathrm{T}'', \sigma_0 \cup \sigma_1}$$

To type check the **do** expression e_0, (T-assume-unsound) takes advantage of the fact that expressions e and e' satisfy the relation $\mathbb{R}$, *i.e.*, in the third condition of the rule, we strengthen the current Φ with $\sigma\ \mathbb{R}\ \sigma'$. The (T-safe) rule in Figure 9 says that the expression type checks iff Φ entails the abstract effect relationship $\mathbb{R}$. As a result, a $\mathbb{SAFE}$ expression whose safety happens to rely on $\sigma\ \mathbb{R}\ \sigma'$ can be verified to be safe by the static system.

Albeit tempting, the rule above is unsound. To illustrate, consider the safe parallelism discipline in the following example, *i.e.*, we instantiate the $\mathbb{R}$ relation with noninterference relation # and the $\mathbb{SAFE}$ expression with parallel expression ||.

Typing: $\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T},\sigma$

(T-Bool) $\Phi;\Gamma \vdash \mathrm{b} : \mathbf{Bool},\emptyset$

(T-var) $$\frac{\mathrm{T} \preceq \Gamma(x)}{\Phi;\Gamma \vdash \mathrm{x} : \mathrm{T},\emptyset}$$

(T-let) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T},\sigma \qquad \Phi;\Gamma,\mathrm{x} \mapsto \mathit{Gen}(\Gamma,\sigma)(\mathrm{T}) \vdash \mathrm{e}' : \mathrm{T}',\sigma'}{\Phi;\Gamma \vdash \mathbf{let}\ \mathrm{x} = \mathrm{e}\ \mathbf{in}\ \mathrm{e}' : \mathrm{T}',\sigma \cup \sigma'}$$

(T-sub) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T},\sigma \qquad \Phi \vdash \mathrm{T} \preceq: \mathrm{T}' \qquad \Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'}{\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T}',\sigma'}$$

(T-abs) $$\frac{\emptyset;\Gamma,\mathrm{x} \mapsto \mathrm{T} \vdash \mathrm{e} : \mathrm{T}',\sigma}{\Phi;\Gamma \vdash \lambda \mathrm{x} : \mathrm{T}.\mathrm{e} : \mathrm{T} \xrightarrow{\sigma} \mathrm{T}',\emptyset}$$

(T-app) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T} \xrightarrow{\sigma} \mathrm{T}',\sigma' \qquad \Phi;\Gamma \vdash \mathrm{e}' : \mathrm{T},\sigma''}{\Phi;\Gamma \vdash \mathrm{e}\ \mathrm{e}' : \mathrm{T}',\sigma \cup \sigma' \cup \sigma''}$$

(T-ref) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T},\sigma}{\Phi;\Gamma \vdash \mathbf{ref}\ \rho\ \mathrm{T}\ \mathrm{e} : \mathbf{Ref}_\rho\ \mathrm{T},\sigma \cup \mathbf{init}_\rho \mathrm{T}}$$

(T-get) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathbf{Ref}_\rho\ \mathrm{T},\sigma}{\Phi;\Gamma \vdash !\ \mathrm{e} : \mathrm{T},\sigma \cup \mathbf{rd}_\rho \mathrm{T}}$$

(T-set) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathbf{Ref}_\rho\ \mathrm{T},\sigma \qquad \Phi;\Gamma \vdash \mathrm{e}' : \mathrm{T},\sigma'}{\Phi;\Gamma \vdash \mathrm{e} := \mathrm{e}' : \mathrm{T},\sigma \cup \sigma' \cup \mathbf{wr}_\rho \mathrm{T}}$$

(T-if-then-else) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathbf{Bool},\sigma \qquad \Phi;\Gamma \vdash \mathrm{e}_0 : \mathrm{T},\sigma_0 \qquad \Phi;\Gamma \vdash \mathrm{e}_1 : \mathrm{T},\sigma_1}{\Phi;\Gamma \vdash \mathbf{if}\ \mathrm{e}\ \mathbf{then}\ \mathrm{e}_0\ \mathbf{else}\ \mathrm{e}_1 : \mathrm{T},\sigma \cup \sigma_0 \cup \sigma_1}$$

(T-assume) $$\frac{\begin{array}{c}\overline{\mathrm{x}} = \mathit{fv}(\mathrm{e}) \cup \mathit{fv}(\mathrm{e}') \qquad \Gamma(\overline{\mathrm{x}}) = \overline{\tau} \qquad \Phi'' \vdash \mathit{EGen}(\overline{\tau}) \Rightarrow \overline{\tau}' \qquad \Gamma' = \Gamma, \overline{\mathrm{x} \mapsto \tau'} \qquad \Phi' = \Phi, \Phi'' \\ \Phi';\Gamma' \vdash \mathrm{e} : \mathrm{T},\sigma \qquad \Phi';\Gamma' \vdash \mathrm{e}' : \mathrm{T}',\sigma' \\ \Phi',\sigma\ \mathbb{R}\ \sigma';\Gamma' \vdash \mathrm{e}_0 : \mathrm{T}''',\sigma_2 \qquad \Phi \vdash \mathrm{T}''' \uparrow \mathrm{T}'' \qquad \Phi \vdash \sigma_2 \uparrow \sigma_0 \qquad \Phi;\Gamma \vdash \mathrm{e}_1 : \mathrm{T}'',\sigma_1\end{array}}{\Phi;\Gamma \vdash \mathbf{assuming}\ \mathrm{e}\ \mathbb{R}\ \mathrm{e}'\ \mathbf{do}\ \mathrm{e}_0\ \mathbf{else}\ \mathrm{e}_1 : \mathrm{T}'',\sigma_0 \cup \sigma_1}$$

(T-safe) $$\frac{\Phi;\Gamma \vdash \mathrm{e} : \mathrm{T}_0,\sigma_0 \qquad \Phi;\Gamma \vdash \mathrm{e}' : \mathrm{T}_1,\sigma_1 \qquad \Phi \vdash_{ar} \sigma_0\ \mathbb{R}\ \sigma_1 \qquad \mathit{clientT}(\mathrm{T},\sigma,\mathrm{T}_0,\sigma_0,\mathrm{T}_1,\sigma_1)}{\Phi;\Gamma \vdash \mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' : \mathrm{T},\sigma}$$

Figure 9 λ_{ie} Typing Rules.

▶ **Example 4** (Soundness Challenge). In the following example, the variables `x` and `y` have the same static (but different dynamic) type. Thus, the expression `x 3` and `y 3` have the same static effect. Should the parallel expression at line 5 typecheck with the assumption expression at line 4, there would be a data race at runtime.

```
1  let buff = ref 0 in
2   let x = if 1 > 0 then λz. !buff else λz. buff := z in
3    let y = if 0 > 1 then λz. !buff else λz. buff := z in
4     assuming !buff  # x 3
5           do !buff || y 3
6         else !buff  ; x 2
```

In this example, we have an imperative reference `buff`, and two structurally similar but distinct functions `x` and `y`. The code intends to perform parallelization, *i.e.*, `buff || y 3`, line 5. Let us review the types of the variables:

$$\begin{array}{rcl} \texttt{buff} & : & \mathbf{Ref}_\rho\ \mathbf{Int} \\ \texttt{x} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int},\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \\ \texttt{y} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int},\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \end{array}$$

∀ Introduction: **Gen** $Gen(\Gamma, \sigma)(\mathrm{T}) = \forall \vec{g}.\mathrm{T}$ where $\vec{g} = ftv(\mathrm{T}) \backslash (ftv(\Gamma) \cup ftv(\sigma))$

∀ Elimination: $\preceq$ $\mathrm{T}' \preceq \forall \vec{g}.\mathrm{T}$ if $\mathrm{T}' = \theta\mathrm{T}$ for some θ

∃ Introduction: **EGen**

$$
\begin{array}{rcll}
\mathbb{P} & ::= & - \mid \mathbf{Ref}_{\mathbb{PR}}\ \mathrm{T} \mid \mathrm{T} \xrightarrow{\mathbb{PE}} \mathrm{T} \mid \mathrm{T} \xrightarrow{\sigma} \mathbb{P} & \textit{pack context} \\
\mathbb{PE} & ::= & - \mid \overline{\omega}, \mathbb{PE}, \overline{\omega}' \mid acc_{\mathbb{PR}}\ \mathrm{T} \mid acc_{\rho}\ \mathbb{P} & \\
\mathbb{PR} & ::= & - \mid \overline{\zeta}, \mathbb{PR}, \overline{\zeta}' &
\end{array}
$$

$$
\begin{array}{rcll}
EGen(\forall \vec{g}.\mathrm{T}) & \triangleq & \forall \vec{g}.EGenM(\mathrm{T}, \emptyset) & \\
EGenM(\mathbb{PE}[\sigma], \vec{g}) & \triangleq & \exists \varsigma \preceq: \sigma.EGenM(\mathbb{PE}[\varsigma \preceq: \sigma], \vec{g} \cup \{\varsigma\}) & \text{if } \sigma \notin \vec{g}, ftv(\sigma) \subseteq \vec{g}, \varsigma \text{ fresh} \\
EGenM(\mathbb{PR}[\rho], \vec{g}) & \triangleq & \exists \gamma \preceq: \rho.EGenM(\mathbb{PR}[\gamma \preceq: \rho], \vec{g} \cup \{\gamma\}) & \text{if } \rho \notin \vec{g}, ftv(\rho) \subseteq \vec{g}, \gamma \text{ fresh} \\
EGenM(\mathrm{T}, \vec{g}) & \triangleq & \mathrm{T} & \text{if } \sigma \in \vec{g} \text{ for any } \mathrm{T} = \mathbb{PE}[\sigma] \\
 & & & \quad \rho \in \vec{g} \text{ for any } \mathrm{T} = \mathbb{PR}[\rho]
\end{array}
$$

∃ Elimination: $\Rrightarrow$ $\forall \vec{g}.(\theta\Sigma) \vdash \forall \vec{g}.\exists \Sigma.\mathrm{T} \Rightarrow \forall \vec{g}.\theta\mathrm{T}$ for some $\theta \wedge dom(\theta) \subseteq \vec{g}$

Lifting: $\uparrow$

$$
\begin{array}{c}
\Phi \vdash \mathbb{P}[\varsigma \preceq: \sigma] \uparrow \mathbb{P}[\sigma'] \text{ if } \forall \vec{g}.\Sigma \in \Phi, \varsigma \preceq: \sigma \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \sigma \uparrow \sigma' \\
\Phi \vdash \mathbb{P}[\gamma \preceq: \rho] \uparrow \mathbb{P}[\rho'] \text{ if } \forall \vec{g}.\Sigma \in \Phi, \gamma \preceq: \rho \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \rho \uparrow \rho' \\
\Phi \vdash \mathrm{T} \uparrow \mathrm{T} \text{ if } \forall \vec{g}.\Sigma \in \Phi, ftv(\mathrm{T}) \cap (\forall \vec{g}.\Sigma) = \emptyset
\end{array}
$$

$$
\begin{array}{c}
\Phi \vdash \mathbb{PE}[\varsigma \preceq: \sigma] \uparrow \mathbb{PE}[\sigma'] \text{ if } \forall \vec{g}.\Sigma \in \Phi, \varsigma \preceq: \sigma \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \sigma \uparrow \sigma' \\
\Phi \vdash \mathbb{PE}[\gamma \preceq: \rho] \uparrow \mathbb{PE}[\rho'] \text{ if } \forall \vec{g}.\Sigma \in \Phi, \gamma \preceq: \rho \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \rho \uparrow \rho' \\
\Phi \vdash \sigma \uparrow \sigma \text{ if } \forall \vec{g}.\Sigma \in \Phi, ftv(\sigma) \cap (\forall \vec{g}.\Sigma) = \emptyset
\end{array}
$$

$$
\begin{array}{c}
\Phi \vdash \mathbb{PR}[\gamma \preceq: \rho] \uparrow \mathbb{PR}[\rho'] \text{ if } \forall \vec{g}.\Sigma \in \Phi, \gamma \preceq: \rho \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \rho \uparrow \rho' \\
\Phi \vdash \rho \uparrow \rho \text{ if } \forall \vec{g}.\Sigma \in \Phi, ftv(\rho) \cap (\forall \vec{g}.\Sigma) = \emptyset
\end{array}
$$

Figure 10 Definitions for ∀ and ∃ Introduction and Elimination.

and the effects of the expressions:

$$
\begin{array}{rcl}
\texttt{!buff} & : & \{\mathbf{rd}_{\rho}\mathbf{Int}\} \\
\texttt{x 3} & : & \{\mathbf{rd}_{\rho}\mathbf{Int}, \mathbf{wr}_{\rho}\mathbf{Int}\} \\
\texttt{y 3} & : & \{\mathbf{rd}_{\rho}\mathbf{Int}, \mathbf{wr}_{\rho}\mathbf{Int}\}
\end{array}
$$

According to the static system, the types of `x` and `y` are exactly the same. Thus, by the third condition of (T-SAFE), the expression `buff || y 3` on line 5, is well-formed. This is because the **assuming** expression has placed $\{\mathbf{rd}_{\rho}\mathbf{Int}\}$ # $\{\mathbf{rd}_{\rho}\mathbf{Int}, \mathbf{wr}_{\rho}\mathbf{Int}\}$ as an element of the relationship set, after typechecking `buff # x 3` on line 4.

At runtime, the initialization expression of the **let** expressions will be first evaluated before being assigned to the variables (*call-by-value*, details in Section 5). Therefore, `x` becomes `λz.!buff` and `y` becomes `λz.buff:=z` before the **assuming** expression. Informally, we refer to the effect computed at runtime through dynamic typing (*e.g.*, right before the **assuming** expression) as *dynamic effect*, as opposed to the *static effect* computed at compile time. The dynamic types of the variables are:

$$\begin{array}{rcl} \texttt{x} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int}\}} \mathbf{Int} \\ \texttt{y} & : & \mathbf{Int} \xrightarrow{\{\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \end{array}$$

and effects of the relevant expressions are:

$$\begin{array}{rcl} \texttt{!buff} & : & \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{x 3} & : & \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{y 3} & : & \{\mathbf{wr}_\rho\mathbf{Int}\} \end{array}$$

Clearly, the dynamic effect computed for `buff` and that for `x 3` on line 4 do not conflict. Therefore, the **do** expression `buff || y 3` will be evaluated. However, the effects of the expressions `buff` and `y 3` on line 5 do conflict, which causes unsafe parallelism.

The root cause of the problem is that the static and the dynamic system make decisions based on two related but different effects: one with the static effect, and the other with the dynamic effect. A sound type system must be able to differentiate the two.

A Sound Design with Bounded Existentials

The key insight from the discussion above is that the static system must be able to express the *dynamic effect* that the **assuming** expression makes decision upon. Before we move on, let us first state several simple observations:

(i) (Dynamic effect refines static effect) The static effect of an expression e is a conservative approximation of the dynamic effect.

(ii) (Free variables determine effect difference) Improved precision of intensional effect polymorphism is achieved by using the more precise types for the free variables, see *e.g.*, dynamic and static type of the variable `x` in Example 4.

Observation *(i)* indicates the possibility of referring to the dynamic effect as "there exists some effect that is subsumed by the static effect." Observation *(ii)* further suggests that dynamic effect can be computed by treating all free variables existentially: "there exists some type T which is a subtype of the static type T′ for each free variable, to help mimic the type environment while dynamic effect is computed". Bounded existential types provide an ideal vehicle for expressing this intention.

(T-Assume) captures the type checking of an **assuming** expression. We substitute the type of each free variable with (an instance of) its existential counterpart. Let us revisit Example 4, this time with (T-Assume). The free variables of the **assuming** expression at line 4, in Example 4 are `x` and `buff`. The original types of the free variables are:

$$\texttt{buff} \;:\; \mathbf{Ref}_\rho\ \mathbf{Int} \qquad \text{and} \qquad \texttt{x} \;:\; \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int},\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int}$$

The existential types used to type check the **assuming** expressions are:

$$\texttt{buff} \;:\; \mathbf{Ref}_\rho\ \mathbf{Int} \quad \text{and} \quad \texttt{x} \;:\; \exists \varsigma_1 \preceq: \{\mathbf{rd}_\rho\mathbf{Int}\}, \varsigma_2 \preceq: \{\mathbf{wr}_\rho\mathbf{Int}\}.\mathbf{Int} \xrightarrow{\varsigma_1,\varsigma_2} \mathbf{Int}$$

The relationship set is:

$$\Phi = \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int} \tag{1}$$

The effects of the condition expressions are:

$$\texttt{!buff} \;:\; \mathbf{rd}_\rho\mathbf{Int} \qquad \text{and} \qquad \texttt{x 3} \;:\; \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}$$

Parallelism: $\|\|$	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathrm{T}_0 = \mathrm{T}_1) \wedge (\sigma = \sigma_0 \cup \sigma_1)$
Security: **exec**	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathrm{T}_1) \wedge (\sigma = \sigma_1)$
UI: **spawn**	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathbf{void}) \wedge (\sigma = \emptyset)$
Memoization: **lookup**	$clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathrm{T} = \mathrm{T}_1) \wedge (\sigma = \sigma_0)$

Figure 11 Client Implementation of Predicate *clientT*.

To type check the **do** expression, Φ is strengthened as:

$$\Phi' = \{\mathbf{rd}_\rho\mathbf{Int} \# \{\varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}\}, \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}\} \tag{2}$$

When type checking the expression on line 5, `y 3` has effect $\{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\}$. We cannot establish $\vdash_{ar}$ (as Figure 8). A type error is correctly induced against the potential unsafe parallel expression.

Rule (T-ASSUME) first computes the free variables from the two condition expressions, written $\overline{\mathrm{x}} = fv(\mathrm{e}) \cup fv(\mathrm{e}')$. With assumption $\Gamma(\overline{\mathrm{x}}) = \overline{\tau}$, all free variables $\overline{\mathrm{x}}$ are considered for type environment strengthening. It then applies the existential *introduction* function *EGen* to strengthen τ', the bounded existential with the original type τ as the bound. The definition of *EGen* is in Figure 10. It then *eliminates* (or *open*) the existential quantification using $\Rightarrow$. In a nutshell, this predicate $\Phi'' \vdash EGen(\overline{\tau}) \Rightarrow \overline{\tau}'$ introduces an existential type and eliminates it right away (a common strategy in building abstract data types [30]). Subsumption relationship information is placed into the relationship set, $\Phi' = \Phi, \Phi''$. The new environment Γ' has the new types $\overline{\tau}'$ for the free variables, an instantiation of the bounded existential type.

Function *EGen* uses the *EGenM* function to quantify effects and regions. Here, to produce the existential type, function *EGen* maintains the structure of the original type, *e.g.*, if the original type is a function type, it produces a new function type with all *covariant* types/effects/regions quantified. Observe that *contravariant* types/effects/regions are harmless: their dynamic counterpart (which also refines the static one) does not cause soundness problems. To facilitate the quantification (also known as existential introduction or *packed* [30]), three *pack contexts*, $\mathbb{P}$, $\mathbb{PE}$, $\mathbb{PR}$, are defined, representing the contexts to contain a type, an effect, or a region, respectively.

Finally, the type of the **do** expression needs to be *lifted*, weakening types that may potentially contain refreshed generic variables of existential types, through a self-explaining $\uparrow$ definition in Figure 10. For example, the effect computed for the expression `x 3` is $\varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}$. The $\uparrow$ function applies the substitution of $\{\varsigma_1 \mapsto \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \mapsto \mathbf{wr}_\rho\mathbf{Int}\}$ on the precomputed effect and produces static effect $\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}$.

The typing of (T-SAFE) relies on the client function *clientT*. $clientT(\mathrm{T}, \sigma, \mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$ defines the conditions where a safe expression should typecheck, as shown in Figure 11.

5 Dynamic Semantics

This section describes the dynamic semantics of λ_{ie}. The highlight is to support a highly precise notion of effect polymorphism via a *lightweight* notion of dynamic typing, which we call *differential alignment*.

Definitions:

$$
\begin{array}{lcll}
s & ::= & \overline{l \rightarrow_{\langle \rho, \mathrm{T} \rangle} v} & \textit{store} \\
f & ::= & \overline{acc(l)} & \textit{trace} \\
v & ::= & \ldots \mid l & \textit{(extended) values} \\
\mathbb{E} & ::= & - \mid \mathbb{E}\ \mathrm{e} \mid v\ \mathbb{E} \mid \textbf{let}\ \mathrm{x} = \mathbb{E}\ \textbf{in}\ \mathrm{e} \mid \textbf{let}\ \mathrm{x} = v\ \textbf{in}\ \mathbb{E} \mid \textbf{ref}\ \rho\ \mathrm{T}\ \mathbb{E} & \textit{evaluation context} \\
 & & \mid !\mathbb{E} \mid \mathbb{E} := \mathrm{e} \mid v := \mathbb{E} \mid \textbf{if}\ \mathbb{E}\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e} &
\end{array}
$$

Dynamic Typing: $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$

$$\text{(DT-LOC)}\quad \frac{\{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\} \in s}{s; \Phi; \Gamma \vdash_{\mathrm{D}} l : \mathbf{Ref}_{\rho}\ \mathrm{T}, \emptyset}$$

For all other (DT-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of judgment $\Phi; \Gamma \vdash \mathrm{e} : \mathrm{T}, \sigma$ in the latter rule should be substituted with $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$ in the former.

Evaluation relation: $s; \mathrm{e}; f \rightarrow s'; \mathrm{e}'; f'$

$$
\begin{array}{lrll}
(cxt) & s; \mathbb{E}[\mathrm{e}]; f \rightarrow s'; \mathbb{E}[\mathrm{e}']; f, f' & \text{if} & s; \mathrm{e} \Rightarrow s'; \mathrm{e}'; f' \\
(asm) & s; \textbf{assuming}\ \mathrm{e}_1\ \mathbb{R}\ \mathrm{e}_2 \Rightarrow s; \mathrm{e}_0; \emptyset & \text{if} & s; \emptyset; \emptyset \vdash_{\mathrm{D}} \mathrm{e}_i : \mathrm{T}_i, \sigma_i \text{ for } i = 1, 2 \\
 & \textbf{do}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}' & & \text{and } \mathrm{e}_0 = \begin{cases} \mathrm{e} & \text{if } \sigma_1\ \mathbb{R}\ \sigma_2 \\ \mathrm{e}' & \text{otherwise} \end{cases} \\
(safe) & s; \mathbb{SAFE}\ \mathrm{e}\ \mathrm{e}' \Rightarrow clientR(s, \mathrm{e}, \mathrm{e}') & & \\
(set) & s; l := v \Rightarrow s, \{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\}; v; \mathbf{wr}(l) & \text{if} & \{l \mapsto_{\langle \rho, T \rangle} v'\} \in s \\
(ref) & s; \textbf{ref}\ \rho\ \mathrm{T}\ v \Rightarrow s, \{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\}; l; \mathbf{init}(l) & \text{if} & l \textit{ fresh} \\
(get) & s; !l \Rightarrow s; s(l); \mathbf{rd}(l) & & \\
(app) & s; \lambda \mathrm{x} : \mathrm{T}.\mathrm{e}\ v \Rightarrow s; [\mathrm{x} \mapsto v]\mathrm{e}; \emptyset & & \\
(let) & s; \textbf{let}\ \mathrm{x} = v\ \textbf{in}\ \mathrm{e} \Rightarrow s; [\mathrm{x} \mapsto v]\mathrm{e}; \emptyset & & \\
(ifT) & s; \textbf{if}\ \textit{true}\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}' \Rightarrow s; \mathrm{e}; \emptyset & & \\
(ifF) & s; \textbf{if}\ \textit{false}\ \textbf{then}\ \mathrm{e}\ \textbf{else}\ \mathrm{e}' \Rightarrow s; \mathrm{e}'; \emptyset & &
\end{array}
$$

Figure 12 λ_{ie} Operational Semantics.

Operational Semantics Overview

The λ_{ie} runtime configuration consists of a *store* s, the to-be-evaluated expression e, and a *trace* f, defined in Figure 12. The store maps references (or locations) l to values v. In addition to booleans and functions, locations themselves are values as well. Each store cell also records the region ρ and type T information of the reference. A *trace* can informally be viewed as "realized effects," and it is defined as a sequence of accesses to references, with $\mathbf{init}(l)$, $\mathbf{rd}(l)$, and $\mathbf{wr}(l)$, denoting the instantiation, read, and write to location l respectively. Traces are only needed to demonstrate the properties of our language. This structure and its runtime maintenance are unnecessary in a λ_{ie} implementation.

The small-step semantics is defined by relation $s; \mathrm{e}; f \rightarrow s'; \mathrm{e}'; f'$, which says that the evaluation of an expression e with the store s and trace f results in the new expression e', the new store s', and the new trace f'. We use notation $[\mathrm{x} \mapsto v]\mathrm{e}$ to define the substitution of x with v of expression e. We use $\rightarrow^*$ to represent the reflexive and transitive closure of $\rightarrow$.

Dynamic Effect Inspection

Most reduction rules are conventional, except *(asm)* and *(safe)*. The *(asm)* rule captures the essence of the **assuming** expression, which relies on dynamic typing to achieve dynamic effect inspection. Dynamic typing is defined through type derivation $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$, defined in the same figure, which extends static typing with one additional rule for reference value typing.

At runtime, the **assuming** expression retrieves the more precise dynamic effect of expression e_1 and e_2, and checks whether relation $\mathbb{R}$ holds. Observe that at runtime, e_1 and e_2 in the **assuming** expression are not identical to their respective forms when the program is written. Now, the free variables in the static program have been substituted with values, which carry more precise information on types, regions, and effects. This is the root cause why intensional effect polymorphism can achieve higher precision than a purely static effect system.

It should be noted that we evaluate neither e_1 nor e_2 at the evaluation of the **assuming** expression. In other words, λ_{ie} is not an *a posteriori* effect monitoring system.

The reduction of *(safe)* relies on an abstract function *clientR*. $\mathit{clientR}(s, \mathrm{e}, \mathrm{e}')$ computes the runtime configuration after the one-step evaluation of the $\mathbb{SAFE}$ expression. For example, for the information security example (Figure 2), the configuration $s; \mathit{adv.show}(\mathit{this}); \emptyset$ will be the result of the *clientR* function. The abstract treatment of this function allows λ_{ie} to be defined in a highly modular fashion, similar to previous work [28]. We will come back to this topic, especially its impact on soundness, in Sec. 6.

Optimization: Efficient Effect Inspection via Differential Alignment

The reduction system we have introduced so far may not be efficient: it requires full-fledged dynamic typing, which may entail dynamic construction of type derivations to compute the dynamic effects. In this section, we introduce one optimization.

As observed in Section 4.4, the (sub)expressions that do not have free variables will have the same static effects (*i.e.*, computed via static typing) and dynamic effects (*i.e.*, computed via dynamic typing). Our key insight is that, the only "difference" between the two forms of effects for the same expression lies with those introduced by free variables in the expression. As a result, we define a new dynamic effect computation strategy with two steps:

1. At compile time, we compute the static effects of the two expressions used for the effect inspection of each **assuming** expression in the program. In the meantime, we record the type (which contains free type/effect/region variables) of each free variable that appears in these two expressions.
2. At runtime, we "align" the static type of each free variable with the dynamic type associated with the corresponding value that substitutes for that free variable. The alignment will compute a substitution of (static) type/effect/region variables to their dynamic counterparts. The substitution will then be used to substitute the effect we computed in Step 1 to produce the dynamic effect.

For Step 1, we define a transformation from expression e to an *annotated expression* d, defined in Figure 13. The two forms are identical, except that the **assuming** expression in the "annotated expression" now takes the form of **assuming** $(\overrightarrow{\mathrm{x} : \tau})$ $\mathrm{e}_1 : \sigma_1$ $\mathbb{R}$ $\mathrm{e}_2 : \sigma_2$ **do** e **else** e', which records the free variables of expressions e_1 and e_2 and their corresponding static types, denoted as $\overrightarrow{\mathrm{x} : \tau}$. The same expression also records the statically computed effects σ_1 and σ_2 for e_1 and e_2. The free variable computation function *fv* and variable substitution

Abstract Syntax in Optimized λ_{ie}

$$
\begin{array}{rcll}
\mathrm{d} & ::= & v \mid \mathrm{x} \mid \mathrm{d}\ \mathrm{d} \mid \textbf{let}\ \mathrm{x} = \mathrm{d}\ \textbf{in}\ \mathrm{d} \mid \textbf{if}\ \mathrm{d}\ \textbf{then}\ \mathrm{d}\ \textbf{else}\ \mathrm{d} & \textit{annotated expressions} \\
& \mid & \textbf{ref}\ \rho\ \mathrm{T}\ \mathrm{d} \mid\ !\mathrm{d} \mid \mathrm{d}\ \texttt{:=}\ \mathrm{d} & \\
& \mid & \textbf{assuming}\ (\overrightarrow{\mathrm{x} : \tau})\ \mathrm{d} : \sigma\ \mathbb{R}\ \mathrm{d} : \sigma\ \textbf{do}\ \mathrm{d}\ \textbf{else}\ \mathrm{d} \mid \mathbb{SAFE}\ \mathrm{d}\ \mathrm{d} &
\end{array}
$$

Transformation: $\mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}$

$$
\begin{array}{rcll}
\mathrm{x} & \overset{\Phi,\Gamma}{\rightsquigarrow} & \mathrm{x} & \\
\mathrm{e}\ \mathrm{e}' & \overset{\Phi,\Gamma}{\rightsquigarrow} & \mathrm{d}\ \mathrm{d}' & \text{if } \mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}, \mathrm{e}' \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}' \\
& \vdots & & \\
\textbf{assuming}\ \mathrm{e}_1\ \mathbb{R}\ \mathrm{e}_2 & \overset{\Phi,\Gamma}{\rightsquigarrow} & \textbf{assuming}\ (\overrightarrow{\mathrm{x} : \tau})\ \mathrm{d}_1 : \sigma_1\ \mathbb{R}\ \mathrm{d}_2 : \sigma_2 & \text{if } \overline{\mathrm{x}} = \mathit{fv}(\mathrm{e}_1) \cup \mathit{fv}(\mathrm{e}_2),\quad \Gamma(\overline{\mathrm{x}}) = \overline{\tau}' \\
\textbf{do}\ \mathrm{e}_3\ \textbf{else}\ \mathrm{e}_4 & & \textbf{do}\ \mathrm{d}_3\ \textbf{else}\ \mathrm{d}_4 & \Phi'' \vdash \mathit{EGen}(\overline{\tau}') \Rightarrow \overline{\tau}, \\
& & & \Phi' = \Phi, \Phi'' \\
& & & \Phi'; \Gamma, \overline{\mathrm{x} \mapsto \tau} \vdash \mathrm{d}_i : \mathrm{T}_i, \sigma_i \text{for } i = 1,2 \\
& & & \mathrm{e}_i \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}_i \text{ for } i = 1,2,3,4
\end{array}
$$

Operational Semantics in Optimized λ_{ie}: $s; \mathrm{d}; f \rightarrow_O s; \mathrm{d}; f$

$$
\begin{array}{rrcll}
(\mathit{Ocxt}) & s; \mathbb{E}[\mathrm{d}]; f & \rightarrow_O & s'; \mathbb{E}[\mathrm{d}']; f, f' & \text{if } s; \mathrm{d}; f \Rightarrow_O s'; \mathrm{d}'; f' \\
(\mathit{Oasm}) & s; \textbf{assuming}(\overrightarrow{v : \tau}) & \Rightarrow_O & s; \mathrm{d}_0; \emptyset & \text{if } s; \emptyset; \emptyset \vdash_{\mathrm{DO}} \overline{v : \mathrm{T}}, \emptyset \\
& \mathrm{d}_1 : \sigma_1\ \mathbb{R}\ \mathrm{d}_2 : \sigma_2 & & & \text{and } \theta\overline{\tau} = \mathit{Gen}(\emptyset, \emptyset)(\overline{\mathrm{T}}) \\
& \textbf{do}\ \mathrm{d}\ \textbf{else}\ \mathrm{d}' & & & \text{and } \mathrm{d}_0 = \begin{cases} \mathrm{d} & \text{if } \theta\sigma_1\ \mathbb{R}\ \theta\sigma_2 \\ \mathrm{d}' & \text{otherwise} \end{cases}
\end{array}
$$

For all other $\Rightarrow_O$ rules, each is isomorphic to its counterpart $\Rightarrow$ rule, except that every occurrence of metavariable e in the latter rule should be substituted with d in the former.

Figure 13 Optimized λ_{ie} with Differential Alignment.

function are defined for d elements in an analogous fashion as for e elements. We omit these definitions.

Considering all the annotated information is readily available while we perform static typing of the **assuming** expression– as in (T-Assume) – the transformation from expression e to annotated expression d under Φ and Γ, denoted as $\mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}$, is rather predictable, defined in the same Figure.

The most interesting part of our optimized system is its dynamic semantics. Here we define a reduction system $\rightarrow_O$, at the bottom of the same figure. We further use $\rightarrow_O^*$ to represent the reflexive and transitive closure of $\rightarrow_O$. Upon the evaluation of the annotated **assuming** expression, the types associated with the free variables – now substituted with values – are "aligned" with the types associated with the corresponding values. The latter is computed by judgment $s; \Phi; \Gamma \vdash_{\mathrm{DO}} \mathrm{d} : \mathrm{T}, \sigma$, defined as $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$ where $\mathrm{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathrm{d}$. In other words, we only need to dynamically type *values* in the optimized λ_{ie}. The alignment is achieved through the computation of the substitution θ. As we shall see in the next section, such a substitution always exists for well-typed programs.

6 Meta-Theories

In this section, we establish formal properties of λ_{ie}. We first show our type system is sound relative to sound customizations of the client effect systems (Section 6.1). We next present important soundness results for intensional effect polymorphism in Section 6.2, and a soundness and completeness result on differential alignment in Section 6.3. The proofs of these theorems and lemmas can be found in the accompanying technical report [25]. Before we proceed, let us first define two simple definitions that will be used for the rest of the section.

▶ **Definition 5.** [Redex Configuration] We say $< s; \mathrm{e}; f >$ is a redex configuration of program e', written $\mathrm{e}' \trianglerighteq <s, \mathrm{e}, f>$, iff $\emptyset; \mathrm{e}'; \emptyset \rightarrow^* s; \mathbb{E}[\mathrm{e}]; f$.

Next, let us define relation $s \vdash f : \sigma$, which says that dynamic trace f *realizes* static effect σ under store s:

▶ **Definition 6.** [Effect-Trace Consistency] $s \vdash f : \sigma$ holds iff $acc(l) \in f$ implies $acc_\rho \mathrm{T} \in \sigma$ where $\{l \mapsto_{\langle \rho, \mathrm{T} \rangle} v\} \in s$.

6.1 Type Soundness

Our type system leaves the definition of $\mathbb{R}$ and $\mathbb{SAFE}$ e e′ abstract, both in terms of syntax and semantics. As a result, the soundness of our type system is conditioned upon how these definitions are concretized. Now let us explicitly define the sound concretization condition:

▶ **Definition 7** (Sound Client Concretization)**.** We say a λ_{ie} client is sound if under that concretization, the following condition holds: if $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e}_0 : \mathrm{T}_0, \sigma_0$, $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e}_1 : \mathrm{T}_1, \sigma_1$, $clientT(\mathrm{T}, \sigma,$
$\mathrm{T}_0, \sigma_0, \mathrm{T}_1, \sigma_1)$ and $(s', \mathrm{e}, f) = clientR(s, \mathrm{e}_0, \mathrm{e}_1)$, then $\mathrm{s}'; \Phi; \Gamma \vdash_{\mathrm{D}} \mathrm{e} : \mathrm{T}, \sigma$ and $s' \vdash f : \sigma$.

All lemmas and theorems for the rest of this section are implicitly under the assumption that Definition 7 holds, which we do not repeatedly state.

Our soundness proof is constructed through subject reduction and progress:

▶ **Lemma 8** (Type Preservation)**.** *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *and* $s; e; f \rightarrow s'; e'; f'$*, then* $s'; \Phi; \Gamma \vdash_D e' : T', \sigma'$ *and* $T' \preceq: T$ *and* $\sigma' \subseteq \sigma$*.*

▶ **Lemma 9** (Progress)**.** *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *then either* e *is a value, or* $s; e; f \rightarrow s'; e'; f'$ *for some* s'*,* e'*,* f'*.*

▶ **Theorem 10** (Type Soundness)**.** *Given an expression* e*, if* $\emptyset; \emptyset \vdash e : T, \sigma$*, then either the evaluation of* e *diverges, or there exist some* s*,* v*, and* f *such that* $\emptyset; e; \emptyset \rightarrow^* s; v; f$*.*

6.2 Soundness of Intensional Effect Polymorphism

The essence of intensional effect polymorphism lies in the fact that through intensional inspection (dynamic typing at the **assuming** expression), every instance of evaluation of the $\mathbb{SAFE}$ e_0 e_1 expression in the reduction sequence must be "safe," where "safety" is defined through the $\mathbb{R}$ relation concretized by the client language. To be more concrete:

▶ **Definition 11** (Effect-based Soundness of Intensional Effect Polymorphism)**.** We say e is effect-sound iff for any redex configuration such that $\mathrm{e} \trianglerighteq <s, \mathrm{e}', f>$ and $\mathrm{e}' = \mathbb{SAFE}\ \mathrm{e}_0\ \mathrm{e}_1$, it must hold that $s; \emptyset; \emptyset \vdash_{\mathrm{D}} \mathrm{e}_0 : \mathrm{T}_0, \sigma_0$ and $s; \emptyset; \emptyset \vdash_{\mathrm{D}} \mathrm{e}_1 : \mathrm{T}_1, \sigma_1$ and $\sigma_0\ \mathbb{R}\ \sigma_1$.

Effect-based soundness is a corollary of type soundness:

▶ **Corollary 12** (λ_{ie} Effect-based Soundness). *If* $\emptyset;\emptyset \vdash e : T, \sigma$, *then* e *is effect-sound.*

There remains a gap between this property and what one *intuitively* believes the execution of $\mathbb{SAFE}\ \mathrm{e}_0\ \mathrm{e}_1$ is "safe": ultimately, what we hope to enforce is at runtime, the "monitored effect" – *i.e.*, the trace through the evaluation of e_0 and that of e_1 – does not violate what $\mathbb{R}$ represents. The definition above falls short because it relies on the dynamic typing of e_0 and e_1. To rigorously define the more intuitive notion of soundness, let us first introduce a trace-based relation induced from $\mathbb{R}$:

▶ **Definition 13** (Induced Trace Relation). $\mathbb{R}^{\mathcal{TR}}$ is a binary relation defined over traces. We say $\mathbb{R}^{\mathcal{TR}}$ is induced from $\mathbb{R}$ under store s iff $\mathbb{R}^{\mathcal{TR}}$ is the smallest relation such that if $\sigma_1\ \mathbb{R}\ \sigma_2$, then $f_1\ \mathbb{R}^{\mathcal{TR}} f_2$ where $s \vdash f_1 : \sigma_1$ and $s \vdash f_2 : \sigma_2$.

One basic property of our reduction system is the trace sequence is monotonically increasing:

▶ **Lemma 14** (Monotone Traces). *If* $s; e; f \to s'; e'; f'$, *then* $f' = f, f''$ *for some* f''.

Given this, we can now define the more intuitive flavor of soundness over traces:

▶ **Definition 15** (Trace-based Soundness of Intensional Effect Polymorphism). We say e is trace-sound iff for any redex configuration such that $\mathrm{e} \trianglerighteq \lessdot s, \mathrm{e}', f \gtrdot$ and $\mathrm{e}' = \mathbb{SAFE}\ \mathrm{e}_0\ \mathrm{e}_1$, it must hold that for any s_0, e'_0, and f_0 where $s; \mathrm{e}_0; f \to^* s_0; \mathrm{e}'_0; f, f_0$ and any s_1, e'_1, and f_1 where $s; \mathrm{e}_1; f \to^* s_1; \mathrm{e}'_1; f, f_1$, then condition $f_0\ \mathbb{R}^{\mathcal{TR}} f_1$ holds.

To prove trace-based soundness, the crucial property we establish is:

▶ **Lemma 16** (Effect-Trace Consistency Preservation). *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$, $s \vdash f : \sigma$ *and* $s; e; f \to s'; e'; f'$ *then* $s' \vdash f' : \sigma'$.

Finally, we can prove the intuitive notion of soundness of intensional effect polymorphism:

▶ **Theorem 17** (λ_{ie} Trace-Based Soundness). *If* $\emptyset;\emptyset \vdash e : T, \sigma$, *then* e *is trace-sound.*

6.3 Differential Alignment Optimization

In Section 5, we defined an alternative "optimized λ_{ie}" to avoid full-fledged dynamic typing, centering on differential alignment. We now answer several important questions: (1) *static completeness*: every typable program in λ_{ie} has a corresponding program in optimized λ_{ie}. (2) *dynamic completeness*: for every typable program in λ_{ie}, its corresponding program at runtime cannot get stuck due to the failure of finding a differential alignment. (3) *soundness*: for every program in λ_{ie}, its corresponding program in optimized λ_{ie} should behave "predictably" at runtime. We will rigorously define this notion shortly; intuitively, it means that "optimized λ_{ie}" is indeed an *optimization* of λ_{ie}, *i.e.*, without altering the results computed by the latter.

Optimization static completeness is a simple property of $\overset{\Phi,\Gamma}{\rightsquigarrow}$:

▶ **Theorem 18** (Static Completeness of Optimization). *For any* e *such that* $\Phi;\Gamma \vdash e : T, \sigma$, *there exists* d *such that* $e \overset{\Phi,\Gamma}{\rightsquigarrow} d$.

To correlate the dynamic behaviors of λ_{ie} and optimized λ_{ie}, first recall that the $\rightarrow$ reduction system and $\rightarrow_O$ reduction system are identical, except for how the **assuming** expression is reduced. The progress of *(Oasm)* relies on the existence of substitution θ that aligns the dynamic type associated with values and the static type. Dynamic completeness of differential alignment thus can be viewed as the "correspondence of progress" for the two reduction systems to reduce the corresponding **assuming** expressions. This is indeed the case, which can be generally captured by the following theorem:

▶ **Theorem 19** (Dynamic Completeness of Optimization). *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *and* $e \overset{\emptyset,\emptyset}{\rightsquigarrow} d$*, then given some* s *and* f*, the following two are equivalent:*

- *there exists some* s'*,* e' *and* f' *such that* $s; e; f \rightarrow s'; e', f'$*.*
- *there exists some* s''*,* d' *and* f'' *such that* $s; d; f \rightarrow_O s''; d', f''$*.*

Finally, we wish to study soundness. The most important insight is that the transformation relation $\overset{\Phi,\Gamma}{\rightsquigarrow}$ can be preserved through the corresponding reductions of λ_{ie} and optimized λ_{ie}. In other words, one can view the reduction of optimized λ_{ie} as a *simulation* of λ_{ie}:

▶ **Lemma 20** ($\rightarrow_O$ Simulates $\rightarrow$ with $\overset{\Phi,\Gamma}{\rightsquigarrow}$ Preservation). *If* $s; \Phi; \Gamma \vdash_D e : T, \sigma$ *and* $e \overset{\emptyset,\emptyset}{\rightsquigarrow} d$ *and* $s; e; f \rightarrow s'; e', f'$ *and* $s; d; f \rightarrow_O s''; d', f''$*, then* $s' = s''$*, and* $f' = f''$*, and* $e' \overset{\emptyset,\emptyset}{\rightsquigarrow} d'$*.*

Finally, let us state our soundness of differential alignment:

▶ **Theorem 21** (Soundness of Optimization). *Given some expression* e *such that* $\emptyset; \emptyset \vdash e : T, \sigma$*, and* $e \overset{\emptyset,\emptyset}{\rightsquigarrow} d$ *then*

- *there exists a reduction sequence such that* $\emptyset; e; \emptyset \rightarrow^* s; v; f$ *iff there exists a reduction sequence such that* $\emptyset; d; \emptyset \rightarrow_O^* s; v; f$*.*
- *there exists a reduction sequence such that the evaluation of* e *diverges according to* $\rightarrow$ *iff there exists a reduction sequence such that the evaluation of* d *diverges according to* $\rightarrow_O$*.*

Observe that we are careful by not stating the two reduction systems must diverge at the same time, or reduce to the same value at the same time. That would be unrealistic if the client instantiations of our calculus introduce non-determinism.

7 Related Work

Static type-and-effect systems are well-explored. Earlier work includes Lucassen [27], and Talpin *et al.* [36], and more recent examples such as Marino *et al.* [28], Task Types [21], Bocchino *et al.* [8] and Rytz *et al.* [32]. There are well-known language design ideas to improve the precision and expressiveness of static type systems, and many may potentially be applied to effect reasoning, such as flow-sensitive types [15], typestates [35] and conditional types [4]. Classic program analysis techniques such as polymorphic type inference, nCFA [33], CPA [3], context-sensitive, flow-sensitive, and path-sensitive analyses, are good candidates for effect reasoning of programs written in existing languages. For example, effect systems can gain more precision by incorporating control flow analysis (nCFA) [33] which provides precise call-site information [23].

Bañados *et al.* [5] developed a gradual effect (GE) type system based on gradual typing [34], by extending Marino *et al.* [28] with `?` ("unknown") types. As a gradual typing system, GE excels in scenarios such as prototyping. The system is also unique in its insight by viewing `?` type concretization as an abstract interpretation problem. Our work shares the high-level philosophy of GE – mixing static typing and dynamic typing for effect reasoning –

but the two systems are orthogonal in approaches. For example, GE programs may run into runtime type errors, whereas our programs do not. Foundationally, the power of intensional effect polymorphism lies upon how parametric polymorphism and intensional type analysis interact – a System F framework on the famous lambda cube – whereas frameworks based on gradual typing are not. Other than gradual typing, other solutions to mix static typing and dynamic typing include the `Dynamic` type [1], soft typing [10] and Hybrid Type Checking [14]. From the perspective of the lambda cube, their expressiveness is on par with gradual typing. Previous work, *e.g.*, Heumann *et al.* [20] and Treichler *et al.* [38], relies on dynamic effects for safe concurrency. Our system is more general: it can not only support safe concurrency as shown in Section 2.1, but also other important application domains such as information security, consistent UI access and program optimization.

Intensional type analysis by Harper and Morrisett [19] is a framework with many extensions (*e.g.*, [12]). We apply it in the context of effect reasoning, and the intentionality in our system is achieved through dynamic typing, instead of `typecase`-style inspection on polymorphic types. To the best of our knowledge, our system is the first hybrid effect type system built on top of the intensional type analysis.

Existential types are commonly used for type abstraction and information hiding. They are also suggested [19, 31] to capture the notion of `Dynamic` type [1]. Our use of existential types are closer to the latter application, except that we aim to differentiate (and connect) the types at compile time and the types at runtime, instead of pessimistically viewing the former as `Dynamic`. We are unaware of the use of *bounded* existential types to connect the two type representations.

Effect systems are an important reasoning aid with many applications. For example, beyond the application domains we described in Section 2, they are also known to be useful for safe dynamic updating [29] and checked exceptions [24, 6].

8 Conclusion

In this paper, we develop a new foundation for type-and-effect systems, where static effect reasoning is coupled with intensional effect analysis powered by dynamic typing. We describe how a precise, sound, and efficient hybrid reasoning system can be constructed, and demonstrate its applications in concurrent programming, information security, UI access, and memoization.

Acknowledgement. We would like to thank all the anonymous reviewers for their insightful comments. We thank Mehdi Bagherzadeh and members of the Panini language team for comments on initial version of this draft and discussion. Yuheng Long would like to thank the members of his PhD POS committee for constructive comments and suggestions.

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

3 Ole Agesen. *Concrete type inference: delivering object-oriented applications.* PhD thesis, Stanford University, Stanford, CA, USA, 1996.

4 Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 1994.

5 Felipe Bañados, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming*, 2014.

6 Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the international workshop on Types in languages design and implementation*, 2007.

7 Guy E. Blelloch. Prefix sums and their applications.

8 Robert L. Bocchino and Vikram 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.

9 Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

10 Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

11 Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Proceedings of Conference on Programming Language Design and Implementation*, 2009.

12 Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the International Conference on Functional Programming*, 1998.

13 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1982.

14 Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

15 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

16 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for controlling UI object access. In *the European Conference on Object-Oriented Programming*, 2013.

17 Aaron Greenhouse and John Boyland. An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, 1999.

18 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2012.

19 Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, 1995.

20 Stephen T. Heumann, Vikram S. Adve, and Shengjie Wang. The tasks with effects model for safe concurrency. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2013.

21 Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *the conference on Object-oriented programming, systems, languages, and applications*, 2010.

22 B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12(2):1–79, 1977.

23 Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *the International Conference on Compiler Construction*, 2005.

24 Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2), 2000.

25 Yuheng Long, Yu David Liu, and Hridesh Rajan. Hybrid Effect Checking. Technical report, Iowa State University, 2014.

26 Yuheng Long, Sean L. Mooney, Tyler Sondag, and Hridesh Rajan. Implicit invocation meets safe, implicit concurrency. In *Ninth International Conference on Generative Programming and Component Engineering*, 2010.

27 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.

28 Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, 2009.

29 Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008.

30 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

31 Andreas Rossberg. Dynamic translucency with abstraction kinds and higher-order coercions. *Electron. Notes Theor. Comput. Sci.*, 218:313–336, 2008.

32 Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

33 Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

34 Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

35 R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.

36 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2), 1994.

37 Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1), 1990.

38 Sean Treichler, Michael Bauer, and Alex 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.