A Type-and-Effect System for Asynchronous, Typed Events

Yuheng Long and Hridesh Rajan
Iowa State University, USA
{csgzlong,hridesh}@iastate.edu

Abstract

Implicit concurrency between handlers is important for event driven systems because it helps simultaneously promote modularity and scalability. Knowing the side-effect of the handlers is critical in these systems to avoid concurrency hazards such as data races. As event systems are dynamic because statically known and unknown handlers can register at almost any time during program execution, static effect analyses must reconcile over competing goals such as precision, soundness and modularity. We recently developed *asynchronous, typed events*, a system that can analyze the effects of the handlers at runtime. This mechanism utilizes runtime information to enable precise effect computation and greatly improves concurrency between handlers.

In this paper, we present the formal underpinnings of asynchronous, typed events, and examine the concurrency safety properties it provides. The technical innovations of our system include a novel effect system to soundly approximate the dynamism introduced by runtime handlers registration, a static analysis to precompute the effects and a dynamic analysis that uses the precomputed effects to improve concurrency. Our design simplifies modular concurrency reasoning and avoids concurrency hazards.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques — Modules and interfaces; D.3.3 [*Programming Languages*]: Language Constructs and Features — Concurrent programming structures

Keywords Implicit concurrency, type-and-effect system

1. Introduction

Event-driven systems, or implicit invocation (II) systems, are popular because of their flexibility and modularity benefits [12, 27, 31, 36]. In these systems, there are two major sets of modules: subjects and handlers. Subjects dynamically announce events, and handlers are implicitly invoked when these events are announced.

Exposing concurrency between handlers in II systems is important because it helps improve responsiveness and scalability [24, 33], but producing it remains a challenge fraught with perils such as data races [32, 33]. Data races, which compromise concurrency safety, happen when two concurrent handlers access the same memory location and at least one of them is a write [14]. A type-and-effect system, or effect system for short, may help understand and avoid these problems because it provides information that encodes and approximates the memory accesses of the handlers [26, 37].

Improving the precision of purely static effect systems is a worthy direction, but looking forward, we believe that they are unlikely to benefit II systems. Our belief is shaped by two insights. First, the configuration of handlers is statically unknown because the decoupling mechanism in II systems allows handlers to be *oblivious* [31] and typically, II systems allow handlers to be dynamically registered [32]. Second, taking the effects of all handlers as an approximation for effect analysis could be over-conservative, as a handler will not execute until after it has registered with the announced event. Therefore, considering the effect of a handler before it registers can be imprecise.

1.1 Background

Recently, we developed *asynchronous, typed events* [24], ATE in short, a system that can analyze the effects of the handlers at runtime to compensate for the conservativeness of static effect analyses. It greatly improves implicit concurrency [24]. To illustrate, consider the following example:

EXAMPLE 1.1 (Effect inspection for implicit concurrency). *In the following program, we would like to decide whether the handlers of the event* `Ev` *can be run concurrently, when* `Ev` *is announced at line 3. In ATE, the announcement at line 15 triggers the event handling mechanism, which will run the two handlers* `r1` *and* `r2` *concurrently. The event announcement at line 17 will run* `r1` *and* `r2` *concurrently and, upon termination, execute* `w`*. Concurrency is improved since* `r1` *and* `r2` *are run concurrently. Concurrency safety is preserved because the conflicting handler* `w` *will not run until* `r1` *and* `r2` *are done.*

```
---------- Server ----------
1 event Ev { Number i; }
2 class Number { int val; }
3 class S { void s(Number k) { announce Ev(k); } }
---------- Client ----------
4 class Read {
5   void reg() { register this.r with Ev; }
6   int r(Number i) { return i.val; } // read effect
7 }
8 class Write {
9   void reg() { register this.w with Ev; }
10  void w(Number i) { i.val = 1; }   // write effect
11 }
12 S s = new S(); Number k = new Number();
13 Read r1 = new Read(); Read r2 = new Read();
14 r1.reg(); r2.reg();
15 s.s(k);                              // line 3, announce
16 Writer w = new Write(); w.reg();
17 s.s(k);                              // line 3, announce
```

Static effect systems The (tricky) concurrency decision of whether to run the handlers concurrently at line 3 depends on the handler configuration, *i.e.* which handlers are registered and in what order, and whether their effects conflict. Due to the following reasons, a static effect system is likely to execute all the handlers sequentially. First, the handler configuration may not be known when the announce expression at line 3 is compiled, due to the decoupling offered by the event type `Ev` [31]. Second, even if all the handlers are known (probably using a whole program analysis, a nemesis for modularity), using the effects of all the handlers as an approximation could be overly conservative for choosing between a concurrent execution and a serial execution. For example, the handlers of the event `Ev` are `r1`, `r2` and `w`. The effects of both `r1` and `r2` are reading `i.val` and the effect of `w` is writing `i.val`, creating a read-write conflict [14]. Therefore, a serial execution has to be used at line 3.

Asynchronous, typed events The serial execution could be over-conservative because at line 15, the conflicting handler `w` has not registered yet and nonconflicting handlers `r1` and `r2` can be run concurrently. The root cause of the conservativeness is that the concurrency decision depends on the handler configuration, which is not known until runtime.

This is precisely where ATE is more effective compared to existing techniques. It can analyze the effects of the handlers at runtime, which enables precise effect computation.

Internally, ATE maintains a *handler queue* for each event. The queue will be mutated and expanded with handler registrations. For instance, before the announcement at line 15, the handler queue has two handlers, `r1` and `r2`. Upon announcement, through dynamic typing, ATE computes the effects of each handler in the queue — gaining highly precise runtime information. Concurrency decisions are guided by these effects, *i.e.*, handlers are run concurrently if the effects do not conflict, *e.g.*, `r1` and `r2`, or else sequentially, *e.g.*, `r1` and `w`. Concurrency is improved with no loss of soundness, as `r1` and `r2` are run concurrently and conflicting handler `w` will not run until `r1` and `r2` are done.

1.2 This Paper: Technical Highlights

Our previous work [24] aims at providing an empirical performance evaluation of ATE's implementation. It has already provided some evidence that ATE is very useful, and can be implemented efficiently to achieve concurrency benefits and has low overhead. However, it did not address important questions about the semantics and the safety properties of this system. In particular, questions relating to the safe concurrency and modularity benefits were not addressed. The purpose of this paper is to address these questions rigorously and provide precise answers.

It turns out that it is challenging to integrate dynamic typing into an event driven system that allows handlers to register dynamically. We now explain the technical difficulties.

Intensional effect inspection At the highest level, ATE shares the philosophy with other dynamic effect systems [16, 23]. However, these systems may not improve the concurrency for II systems where handlers can register dynamically. The reason is that they do not inspect *mutable data structures* (*i.e.*, the mutable handler queue) and thus miss concurrency opportunities. Mechanically extending these systems could have undesirable consequences, because of the long-standing problem in type-and-effect systems: reasoning about polymorphic mutable data structures is notoriously difficult [29, 37]. Consider the following example:

EXAMPLE 1.2 (Post inspection modification). *In the example below, right before the event (`Ev`) announcement at line 15, there are two handlers, an instance `v` of `Subtlety` (line 13) and an instance `r` of `Read` (line 14), in the queue for `Ev`. It may be tempting to conclude that `r` and `v` can be run concurrently, because `v` (lines 9-10) does not have any (direct) read/write effect. A careful reader may say that `v` will* ***announce*** *`Ee` and its handlers could conflict with `r`. Observe that the queue for `Ee` is still empty. Thus, no registered handler interferes with `r`. So intuitively, it is "safe".*

```
---------- Client2 ----------
1 event Ee { Number i; }
2 class Hide {
3   void reg() { register this.h with Ee; }
4   void h(Number i) { i.val = 1; }          // write effect
5 }
6 class Subtlety {
7   void reg() { register this.v with Ev; }
8   void v(Number i) {
9     Hide h = new Hide().reg(); // register hidden handler
10    announce Ee(i);
11  }
12 }
13 Subtlety v = new Subtlety().reg();
14 Read r = new Read().reg();
15 new S().s(new Number());
```

Although tempting, the intuition is unsound. When `v` executes, before announcing `Ee` at line 10, it registers a handler `h` (line 9) for `Ee` — dynamically modifying the mutable handlers queue. The **announce** will now execute `h`, which conflicts with `r`, causing unsafe parallelism. This contradicts our belief that there is no conflicting handler.

The root cause of the problem is unsound reasoning about mutable handler queues and their modification post runtime effect inspection.

To solve the problem, ATE introduces two kinds of effects, namely **reg** for the **register** expression and **ann** for the **announce** expression. The **ann** effect approximates the effects of potential handlers for the to-be-announced event. The **reg** effect captures the modification of handler queues and comes with the (latent) effects [37] of the to-be-register handler, and is the effect incurred when the handler is invoked, *i.e.*, at event announcements. When these two effects combine, the (latent) effects of potential handlers will also be included (detailed in §3), *e.g.*, the combination of the effects of line 9 and line 10 will include the effects of `h`, *i.e.*, the effects of `v` include the **reg**, **ann** and the write effect from `h`. Now, the effects of `v` and `r` conflict and ATE runs them serially, which is desirable for concurrency safety.

Here ATE's static and dynamic systems interact in interesting ways. Dynamic typing provides precise effects — exploiting runtime information — allowing more concurrency opportunities. Static typing precomputes the effects of the handlers — avoiding potential expensive effect computation at runtime — and soundly captures the dynamic modification of the queues, via the **reg** and **ann** effects, which is good news for reasoning about mutable queues.

Modular reasoning Next, we show an important modularity benefit of ATE's design. Modular reasoning about concurrency could be challenging, due to the well known *pervasive interference* problem [3, 20, 39], defined below.

DEFINITION 1.3. *[Pervasive interference] Pervasive interference in a concurrent II program means that between* any two consecutive expressions *of a handler* `h`*, interleaving expressions of another handler could change the states of* `h` *and influence* `h`*'s subsequent behavior.*

To illustrate, consider the following program. The handler `addThenAnnounce` increases the input `Number` by 1 (line 3, using the standard read-increase-set expressions [39]) and announces an event with the modified `Number`.

EXAMPLE 1.4 (Low interference density). *Any two consecutive reads of the variable* `j` *at line 3 could result in two completely different integers because of the potential interference of other handlers. This problem is manifested by adding the* interference points *(*$^{\alpha}$*) to the source program.*

```
1 int j;
2 void addThenAnnounce (Number i) {
3   α j = iα.val; j α= jα + 1; i.val α= α j;
4   announce i Ee(i);
5 }            // the less interference points, the better
```

*In ATE, the number of interference points is 1 (*i*), instead of 7 (*$^{\alpha}$ *and* i*), i.e., between every consecutive expressions. Code that lies within any pair of interference points is a* transaction *or an* atomic *block and thus can be reasoned about sequentially [39]. With ATE, programmers reap the benefits of atomicity when reasoning about handlers.*

In a naive extension of an II language with concurrency but without safety guarantees, programmers must consider all other handlers to determine whether their interleavings would be harmful at every program point. This is illustrated by all $^{\alpha}$ interference points which show global reasoning is required to analyze this program, rather than the modular reasoning we desire. Thanks to the concurrency safety guarantees, ATE controls the interference points i to only after the **announce** expression, instead of every expression.

1.3 Contributions

In summary, this paper makes the following contributions:

- It formalizes a novel effect system where effects of handlers are inspected at runtime to improve concurrency.
- It shows the subtleties resulting from the precise handlers' effect inspection and develops a sound type system and operational semantics to tackle the problem.
- It proves that our system provides sequential semantics [6], even though it is implicitly concurrent.
- It proves that our system can be reasoned about modularly and atomically.

Roadmap: §2 presents ATE's abstract syntax, §3 presents its type system, §4 presents its small-step operational semantics, and §5 describes and proves key properties of our approach. Finally, §6 surveys related work, and §7 concludes.

1.4 Examples

Before we proceed, let us demonstrate the applicability of ATE in reasoning about refactoring mining and static program error detection. Our previous work [24] did not show the benefits of using a dynamic effect analysis in II systems. These examples show the benefits of implicit concurrency between handlers and the power of the dynamic effect analysis in reaping implicit concurrency.

Refactoring crawler This tool uses crawler handlers to mine refactorings, shown in Figure 1, such as renaming, method pullup and pushdown, and changes of method signatures between software versions [10]. The driver class reads and parses two versions of the software, at line 4, and starts the mining processes by announcing the event `Ev`, which in turn will run the registered crawlers.

The tool provides several predefined crawlers and allows client users to implement their own crawlers. It also allows users to selectively register a subset of the crawlers.

It is desirable to execute the crawlers concurrently to detect refactorings in ultra-large-scale software repositories [10, 11]. The tricky problem is that the crawlers may not be known to the tool when it is developed because client users can provide their own crawlers. Also, crawlers may conflict with each other, *e.g.*, the move method (`Move`)

```
- - - - - - - - - - - - - - Library - - - - - - - - - - - - - -
1  event Ev { AST s; AST d; }
2  class Detect {
3    static AST p = null; static AST m = null;
4    void main() {
5      AST s = /* ... */; AST d = /* ... */;
6      announce Ev(s, d);
7    }
8  }
- - - - - - - - - - - - - - Client - - - - - - - - - - - - - -
9  class Move {
10   void reg() { register this.detect with Ev; }
11   void find(AST s, AST d) {
12     if (/* s is move refactoring of d */
13       && Detect.p == null) m = d;
14   }
15 }
16 class Pull {
17   void reg() { register this.detect with Ev; }
18   void find(AST s, AST d) {
19     if (/* s is pull refactoring of d */) Detect.p = d;
20   }
21 }
22 if (random()) new Move().reg();
23 if (random()) new Pull().reg();
```

Figure 1. Refactoring mining in ATE [10].

crawler checks whether a method is classified as a refactoring by some other (conflicting) crawlers (`Pull`). Worse still, crawlers may or may not be invoked depending on user inputs (lines 22-23), which are unknown until runtime.

This is exactly where ATE shines. Our scheduler is able to solve the problem by leveraging the runtime handler information to exploit the concurrency between nonconflicting crawlers, *e.g.*, at line 6, ATE inspects the effects of the handlers and executes nonconflicting handlers concurrently.

FindBugs This famous library [17] uses detector handlers to find bugs. It provides several predefined detectors but also allows clients to implement their custom detectors. This library should be able to leverage parallelism, at line 5, because most of the detectors do not have conflicting effects.

However, the library may not know what the detectors are, nor their effects. These unknown detectors could be provided by client users. There are also predefined detectors that have conflicting effects. For example, the predefined detectors `FindNoSideEffectMethods` and `FindUselessObjects` conflict with each other. One of the steps in the `FindUselessObjects` detector is to check whether a method invocation expression has side effects, using the results produced by `FindNoSideEffectMethods`.

With dynamic typing, ATE inspects the effects of the registered handlers to enable precise reasoning at runtime. Potential concurrency is precisely exploited by scheduling the nonconflicting detectors.

2.A Calculus with Asynchronous Typed Events

The abstract syntax of our calculus that supports asynchronous, typed events is defined in Figure 3. Our calculus is built on top of an imperative object-oriented calculus, and

```
- - - - - - - - - - - - - - Library - - - - - - - - - - - - - -
1  event Ev { Class c; }
2  class FindBugs {
3    void main() {
4      Class c = /* ... */;
5      announce Ev(c);
6    }
7  }
- - - - - - - - - - - - - - Client - - - - - - - - - - - - - -
8  class FindNoSideEffectMethods {
9    static Method meth;
10   void reg() { register this.detect with Ev; }
11   void detect(Class c) {
12     if (/* c.m no side effect */) meth = /* c.m */;
13   }
14 }
15 class FindUselessObjects {
16   void reg() { register this.detect with Ev; }
17   void detect(Class c) {
18     if (c.m == FindNoSideEffectMethods.meth) /* ... */
19   }
20 }
```

Figure 2. FindBugs in ATE [17].

Ptolemy [8, 31]. Key language features are highlighted in blue, which support safe implicit concurrency, and in red, which are challenging for a concurrent II language.

prog	::=	$\overline{decl}\ e$	program
decl	::=	**class** c **extends** d $\{\overline{fld}\ \ \overline{meth}\}$	class
	|	**event** p $\{\overline{form}\}$	event
fld	::=	$c\ f$ **in** ω	field
meth	::=	$c\ m\ (\overline{form})\ \{e\}$	method
t	::=	c | **void**	type
form	::=	$c\ x$, **where** $x \neq$ **this**	parameter
e	::=	$form{=}e; e$ | x | **null** | $e.m\ (\overline{e})$	expression
	|	$e.f$ | $e.f{=}e$ | **new** $c\ ()$	reference
	|	**yield** e	cooperation
	|	**register** **this**$.m$ **with** p	registration
	|	**announce** $p\ (\overline{e})$	announcement

where	c, d	$\in$ $\mathcal{C}$,	the set of class names
	p	$\in$ $\mathcal{P}$,	the set of event names
	f	$\in$ $\mathcal{F}$,	the set of field names
	m	$\in$ $\mathcal{M}$,	the set of method names
	x	$\in$ $\{\textbf{this}\} \cup \mathcal{V}$,	the set of variable names
	ω	$\in$ $\mathcal{R}$,	the set of region names

Figure 3. ATE's abstract syntax. Throughout the paper, notation $\overline{\bullet}$ represents a sequence of $\bullet$ elements.

2.1 Expressions

The syntax includes conventional OO expressions. The highlight of ATE is a few interconnected features:

Dynamic event registration The **register** expression registers handlers with events dynamically (*e.g.*, line 5 in Figure 1.1). As shown in Example 1.2, reasoning about the concurrency safety of an II language with dynamic event registration is challenging. The tricky problem is that the concur-

rency safety depends on the configuration of the handlers, which is not known until event registration at runtime. The registration-time specialization in §4 solves this problem.

Implicit concurrency via event announcement The **announce** expression is the source of implicit concurrency. At runtime, it inspects the effects of each handler and schedules nonconflicting handlers to run concurrently. Two handlers may interfere, referred to as *conflicting handlers*, if their effects access the same memory location and at least one of the accesses is a write [14]. The runtime manages the details of concurrency to relieve programmers from the burden of explicitly managing threads and locks.

Modeling concurrency via cooperative handlers To model concurrency and rigorously prove the properties of ATE, we introduce the **yield** expression to simulate *cooperative handlers* [1, 39]. It may not be used in source programs but serves as an intermediate expression in the semantics (§4), which is used to allow other handlers to run, *i.e.*, a handler can explicitly yield control to other handlers.

The introduction of cooperative handlers could complicate modular reasoning due to the well known *pervasive interference* problem [3, 20, 39] (see Example 1.4). This problem is manifested by adding the **yield** expression (shown as $^{\alpha}$ in Example 1.4), referred to as *interference points*, to the source program. We will show in §5.5 that ATE controls and limits the interference points to only after the **announce** expression, instead of every expression.

2.2 Declarations

A program consists of a sequence of declarations followed by an expression, which can be thought of as the body of a "main" method.

The event type (`event`) declaration facilitates the implicit invocation design style [7, 12, 27, 31, 35], whose intention is to provide a named abstraction for a set of events.

Class declarations are standard except that each field is associated with a *region name* [15, 26, 37], a common way of abstracting memory locations for effect systems to reason about memory accesses. For the examples where regions are not explicitly annotated, different region names suffice.

3. Type and Static Effect Computation

We now describe ATE's type system, which computes precise effects for handlers. The dynamic semantics (§4) will use these effects to determine a safe order for handler invocation and to improve concurrency. The highlight is new effects to approximate the modification of handler queues.

3.1 Effects Reasoning for Mutable Handler Queue

Compared with previous work on static effect reasonings [6, 26, 37], effects of handlers are constantly changing in event based systems [25] (Example 1.1 and 1.2), due to runtime event registrations. The handlers of an event are statically unknown. To tackle the problem, ATE introduces two new effects, the *announce* and *register* effects, expressed as **ann** and **reg**. An **ann** effect serves as a place holder for the concrete effects of zero or more registered handlers and is made concrete during handler registration at runtime (§4).

3.2 Type and Effect Attributes, and Effect Interference

The type attributes used by the type system are defined in Figure 4. The type attributes for expressions are represented as (t, σ): the type t of an expression and its effect set σ.

θ	::=	OK	*decl type*
	|	$\overline{t} \xrightarrow{\sigma} t$ **in** c	*method type*
	|	t, σ	*expression type*
σ	::=	$\overline{\epsilon}$	*program effect*
ϵ	::=	**rd** ω	*read effect*
	|	**wr** ω	*write effect*
	|	**ann** p	*announce effect*
	|	**reg** $p\ \sigma$	*register effect*
Π	::=	$\overline{x : t}$	*type environment*

Figure 4. Type-and-effect attributes.

The interference relation is shown in Figure 5. Read effects do not conflict with each other. Write effects conflict with read and write effects accessing the same region. Event registration **register** will modify the event queue to append the new handler and the **announce** expression will read the queue to execute the registered handlers. Similar to read-/write effects, **reg** conflicts with each other and **ann** accessing the same event p.

Noninterfering Effects, $\sigma \sharp \sigma$:

$$\emptyset \sharp \sigma \qquad \frac{\sigma \sharp \sigma'' \quad \sigma' \sharp \sigma''}{(\sigma \cup \sigma') \sharp \sigma''} \qquad \frac{\sigma \sharp \sigma'}{\sigma' \sharp \sigma} \qquad \textbf{rd}\ \omega \sharp \textbf{rd}\ \omega'$$

$$\frac{\omega \neq \omega'}{\textbf{rd}\ \omega \sharp \textbf{wr}\ \omega'} \qquad \frac{\omega \neq \omega'}{\textbf{wr}\ \omega \sharp \textbf{wr}\ \omega'} \qquad (\textbf{reg}\ p\ \sigma / \textbf{ann}\ p) \sharp \textbf{wr}/\textbf{rd}\ \omega$$

$$\frac{p \neq p'}{\textbf{reg}\ p\ \sigma \sharp \textbf{ann}\ p'} \qquad \frac{p \neq p'}{\textbf{reg}\ p\ \sigma \sharp \textbf{reg}\ p'\ \sigma'} \qquad \textbf{ann}\ p \sharp \textbf{ann}\ p'$$

Figure 5. Effect noninterference.

Notations The notation $t' \preccurlyeq t$ means t' is a subtype of t. It is the reflexive-transitive closure of the declared subclass relationships. We state the type checking rules using a fixed class table (list of declarations CT [18]). The typing rules for expressions use a type environment, Π, which is a finite partial mapping from variable names x to types t.

3.3 Expressions

The rules for expressions are rather conventional, shown in Figure 6. Rules (T-GET) and (T-SET) for store operations produce the read and write effects, respectively. We highlight

Typing: $\Pi \vdash e : t, \sigma$

$$\text{(T-YIELD)}\ \frac{\Pi \vdash e : t, \sigma}{\Pi \vdash \textbf{yield}\ e : t, \sigma} \qquad \text{(T-ANNOUNCE)}\ \frac{\textbf{event}\ p\ \{\overline{t\,x}\} \in CT \qquad \forall\, t_i\, x_i \in \overline{t\,x}\ \text{s.t.}\ \Pi \vdash e_i : t_i', \sigma_i\ \wedge\ t_i' \preccurlyeq t_i}{\Pi \vdash \textbf{announce}\ p\ (\overline{e}) : \textbf{void}, \overline{\sigma} \sqcup \textbf{ann}\ p}$$

$$\text{(T-REGISTER)}\ \frac{\Pi \vdash \textbf{this} : t, \emptyset \quad (c, t'', m\ (\overline{t'\,x})\ \{e\}, \sigma) = \mathit{find}(t, m) \qquad \textbf{event}\ p\ \{\overline{t\,x}\} \in CT \qquad \forall t \in \overline{t}\,.\, t' \preccurlyeq t}{\Pi \vdash \textbf{register}\ \textbf{this}.m\ \textbf{with}\ p : t, \textbf{reg}\ p\ \sigma}$$

$$\text{(T-VAR)}\ \frac{\Pi(x) = t}{\Pi \vdash x : t, \emptyset} \qquad \text{(T-NEW)}\ \frac{\mathit{isClass}(c)}{\Pi \vdash \textbf{new}\ c() : c, \emptyset} \qquad \text{(T-NULL)}\ \Pi \vdash \textbf{null} : c, \emptyset$$

$$\text{(T-DEF)}\ \frac{\Pi \vdash e : c', \sigma \qquad c' \preccurlyeq c \qquad \Pi \oplus \{x \mapsto c\} \vdash e' : t, \sigma'}{\Pi \vdash c\,x = e; e' : t, \sigma \sqcup \sigma'}$$

$$\text{(T-GET)}\ \frac{\Pi \vdash e : c, \sigma \quad \mathit{type}(c, f) = (\omega, t)}{\Pi \vdash e.f : t, \sigma \sqcup \textbf{rd}\ \omega} \qquad \text{(T-SET)}\ \frac{\Pi \vdash e : c, \sigma \qquad \mathit{type}(c, f) = (\omega, t) \qquad \Pi \vdash e' : t', \sigma' \qquad t' \preccurlyeq t}{\Pi \vdash e.f = e' : t', \sigma \sqcup \sigma' \sqcup \textbf{wr}\ \omega}$$

$$\text{(T-CALL)}\ \frac{(c_1, t, m\ (\overline{t\,x})\ \{e_{n+1}\}, \sigma) = \mathit{find}(c_0, m) \qquad \Pi \vdash e_0 : c_0, \sigma_0 \qquad (\forall\, t_i\, x_i \in \overline{t\,x},\ \Pi \vdash e_i : t_i', \sigma_i\ \wedge\ t_i' \preccurlyeq t_i)}{\Pi \vdash e_0.m(\overline{e}) : t, \sigma \sqcup \overline{\sigma}}$$

"Latent" Handler Effects and its Realization, $\sigma \sqcup \sigma = \sigma$:

$$\sigma \sqcup \sigma' = \sigma \cup \sigma' \cup \{\epsilon \mid \textbf{ann}\ p \in \sigma \wedge \textbf{reg}\ p\ \sigma'' \in \sigma' \wedge \exists \epsilon \in \sigma''\}$$

Figure 6. Type-and-effect rules.

the interesting rules. The (T-YIELD) says that a **yield** expression has same type and effect as the expression e.

The (T-REGISTER) says that the effect of a register expression is a register effect **reg** associated with the effects of the to-be-register handler, to model handler queue modification, *e.g.*, in Example 1.2, the effect of the expression at line 9 is **reg** `Ee` **wr** ω, where **wr** ω is the effect of `h`.

The (T-ANNOUNCE) says that the effects of the expression are the union of all the parameters' effects plus one announcement effect, **ann**. This effect serves as a place holder which will be used by registration-time specialization in §4 to fill up more precise effect information at runtime.

The communication of the effects from handler registration to a handler invocation is best viewed in the effect operator $\sqcup$ used in the rules and defined at the bottom of Figure 6. Via the **register** expression, the effect of a handler is put inside the **reg** effect while with the (T-ANNOUNCE) and $\sqcup$, this embedded effect is extracted from the **reg** effect to be exercised at the point of announcement; effects flow from the points where handlers are registered to the points where they are invoked, *e.g.*, in Example 1.2, `h` will run as the result of 1) its registration at line 9 and 2) the **announce** at line 10. Therefore, the effects of `v` include the effects of `h`, when combining the effects **reg** `Ee` **wr** ω and **ann** `Ee`. $\sqcup$ can be viewed as a special form of effect set union $\cup$, which will union the effects of its LHS and RHS, a typical way of merging the effects of subexpressions, *e.g.*, (T-GET).

3.4 Top-Level Declarations

The rules for declarations are standard, shown in Figure 7.

Typings for Declarations:

$$\text{(T-EVENT)}\ \frac{\forall (t\,x) \in \overline{t\,x}, \mathit{isClass}(t)}{\vdash \textbf{event}\ p\ \{\overline{t\,x}\} : \text{OK}} \qquad \text{(T-PROGRAM)}\ \frac{\forall \mathit{decl} \in \overline{\mathit{decl}},\ \vdash \mathit{decl} : \text{OK} \qquad \vdash e : t, \sigma}{\vdash \overline{\mathit{decl}}\ e : t, \sigma}$$

$$\text{(T-METHOD)}\ \frac{\mathit{override}(m, c, \overline{t} \xrightarrow{\sigma} t) \qquad \forall t_i\, x_i \in \overline{t\,x},\ \mathit{isClass}(t_i) \qquad \mathit{isType}(t) \qquad (\overline{x:t}, \textbf{this} : c) \vdash e : t', \sigma \qquad t' \preccurlyeq t}{\vdash t\, m(\overline{t\,x})\{e\} : \overline{t} \xrightarrow{\sigma} t\ \textbf{in}\ c}$$

$$\text{(T-CLASS)}\ \frac{\mathit{validF}(\overline{t\,f}, d) \qquad \forall \mathit{meth} \in \overline{\mathit{meth}},\ \vdash \mathit{meth} : t\ \textbf{in}\ c}{\vdash \textbf{class}\ c\ \textbf{extends}\ d\ \{\overline{t\ f\ \textbf{in}\ \omega}\ \ \overline{\mathit{meth}}\} : \text{OK}}$$

Auxiliary Functions:

$\mathit{isClass}(t)$	**if**	$\textbf{class}\ t \ldots \in CT$
$\mathit{isType}(t)$	**if**	$\mathit{isClass}(t) \vee t = \textbf{void}$
$\mathit{validF}(\overline{t\,f}, c)$	**if**	$\forall (t\,f) \in \overline{t\,f}, \mathit{isClass}(t) \wedge f \notin \mathit{dom}(\mathit{flds}(c))$
$\mathit{flds}(c) = fs$	**if**	$\textbf{class}\ c\ \textbf{extends}\ d\ \{\overline{t\ f\ \textbf{in}\ \omega}\ \ldots\} \in CT$
	$\wedge$	$fs = \mathit{flds}(d) \cup \overline{f \mapsto_\omega t}$

Valid Method Overriding: $\mathit{override}(m, c, \overline{t} \xrightarrow{\sigma} t)$

$\mathit{override}(m, c, \overline{t} \xrightarrow{\sigma} t)$ **if** $(c', t, m\ (\overline{t\,x})\ \{e\}, \sigma') = \mathit{find}(c, m) \wedge \sigma \subseteq \sigma'$

Method Lookup: $\mathit{find}(c, m) = (c', t, m(\overline{t\,x})\{e\}, \sigma)$

$$\mathit{find}(c, m) = \begin{cases} (c, t, m(\overline{t\,x})\{e\}, \sigma) & \textbf{if}\ \textbf{class}\ c\ \textbf{extends}\ d\{\ldots \overline{\mathit{meth}}\} \in CT \\ & \wedge\ (t, \sigma, m(\overline{t\,x})\{e\}) \in \overline{\mathit{meth}} \\ \mathit{find}(d, m) & \textbf{otherwise} \end{cases}$$

Type Lookup for Field: $\mathit{type}(c, f) = (\omega, t)$

$$\mathit{type}(c, f) = \begin{cases} (\omega, t) & \textbf{if}\ \textbf{class}\ c\ \textbf{extends}\ d\ \{\overline{\mathit{field}} \ldots\} \in CT \\ & \wedge\ t\ f\ \textbf{in}\ \omega \in \overline{\mathit{field}} \\ \mathit{type}(d, f) & \textbf{otherwise} \end{cases}$$

Figure 7. Type-and-effect rules for top level declarations.

The (T-METHOD) uses the function *override* (Figure 7) to check overriding, which enforces that the effect of an overriding method is a subset of the overridden method [15].

4. Semantics with Effect-Guided Scheduling

Here we give a small-step operational semantics for ATE. The main novelty is to support precise reasoning of the dynamically changing effects of handlers via registration-time specialization, dynamic typing and the integration of the effect system with a scheduling algorithm that produces safe execution, while improving concurrency for II programs.

4.1 Domains

The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 8. A configuration consists of a task queue ψ, a store μ, an event map γ and a *trace* $\hbar$. Each reference cell in μ records an object $c.F$, consisting of a class name c and a field record. A field record $\overline{f \mapsto_\omega v}$ maps field names f to values v in region ω. A value v may either be `null` or a location l. The map γ maps an event p to a configuration. This configuration consists of a (mutable) queue of handlers $\overline{l.m}$ and a boolean flag b, indicating whether the handlers can be run concurrently.

The task queue ψ consists of an ordered list of task configurations $\langle e, id\rangle$. Each task configuration (called simply a task) consists of the task identifier id and an expression e serving as the remaining evaluation to be done for the task.

A trace $\hbar$ is the "realized effects". It is defined as a sequence of accesses to references, with read/write to regions and event registration and announcement. Traces are only needed to demonstrate the soundness (§5), but are unnecessary in the implementation.

ATE uses a call-by-value evaluation strategy. The operator $\oplus$ is an overriding operator for finite functions, *i.e.*, if $\mu' = \mu \oplus \{l \mapsto o\}$, then $\mu'(l') = o$ if $l' = l$, otherwise $\mu'(l') = \mu(l')$. The rest of this section highlights the rules for the expressions **announce**, **register** and **yield**.

4.2 Registration-Time Specialization & Dynamic Typing

The (REG) rule appends the new handler to the mutable queue $p \mapsto \left\langle b, \overline{l.m} + \overline{l.m} \right\rangle$ for event p. Concurrency decisions can now be made because the previously unknown handlers become known. If none of the handlers conflicts, indicated by the flag b, they can be run concurrently. *Dynamic typing* is used to compute the effect of the new handler.

Dynamic typing provides more precise effects because of two reasons: 1) at runtime, the variables of the source expression $\mathtt{e}$ will be substituted with values (*e.g.*, (DEF) and (CALL)), which carries more precise runtime information [23]; and 2) the previously unknown handlers are known (*registration-time specialization*), by inspecting the queue.

Dynamic typing is defined through type derivation $\gamma, \mu, \Pi \vdash_D e : t, \sigma$ in Figure 8, which extends static typing, defined in Figure 6, with one additional rule (γ-l) for reference l value typing. In previous work, effects do not change at runtime. In ATE, the effects could change due to dynamic event registration, *e.g.*, the effects of a subject that may announce `p`, could change, with more handlers registered with `p`. To account for this, dynamic typing inspects the handlers in the event map γ (the (γ-EVENT) rule) and provides precise effects for the **announce** expressions. The (γ-EVENT) checks for event p whether the handlers for p can be run concurrently, and what the effects of all these handlers are. Note that the dynamic typing rule may recurse on the events when checking an **announce** expression and thus the *fixed point* operator is used.

Note that a handler `h` can register (other) handlers `h'` for an event p when handling an event and later announce the event p (*e.g.*, `v` in Example 1.2). The effects of `h` should include the registration, announce effects and the effects of `h'`. This scenario is handled by $\sqcup$ (Figure 6). An alternative sound solution will let the effects of h to conservatively be top, *i.e.*, read/write the entire store [6].

4.3 Event Announcement & Safe Implicit Concurrency

The (ANN) retrieves the handlers registered for the corresponding event p. The dynamic typing used in the **register** provides precise effects and analyzes whether the handlers can be run concurrently. If their effects conflict, each handler has to wait until the completion of the previous registered handler using the expression $\overrightarrow{join}(\overline{id_{i-1}})$ to avoid concurrency errors. Otherwise, the handlers can all be run concurrently. The **announce** waits for its handlers to complete.

Note that if any pair of handlers conflict, the formalism executes the handlers sequentially. A better implementation is possible, *e.g.*, executing nonconflicting handlers concurrently [24] (concurrent read exclusive write) or executing a handler as soon as all its conflicting handlers are done [16], or executing handlers with less effects before handlers with more effects to promote modular reasoning [2]. These schedulings maintain concurrency safety because conflicting handlers can never be run concurrently. There are many scheduling techniques from which our work can learn, but the simplification suffices to illustrate the soundness.

The expression $\overrightarrow{join}(\overline{id})$ can only process after all the tasks $\overline{id}$ are done, *i.e.*, no longer in the queue ψ. The expression $\overrightarrow{join}$ is joining other handlers and known as a *right mover* [14], indicated by the head symbol $\rightarrow$. As interference points only exist after the right mover [39], **announce** is the only interference points in ATE (see Example 1.4).

4.4 Yielding Control & Interference Points

To model concurrency, we use preemptive interleaving [39], like Abadi and Plotkin [1], *i.e.*, the running handlers will relinquish control (interference points) to other handlers at each step (see the (CONT)). We will prove in §5.5 that, in ATE, this preemptive semantics is equivalent to the cooper-

Definitions:

$$
\begin{array}{rcll}
\Sigma & ::= & \langle \psi, \mu, \gamma, \hbar \rangle & \textit{program configuration} \\
\psi & ::= & \overline{\langle e, id \rangle} & \textit{task queue} \\
id & ::= & id.id \mid 0 \mid 1 \mid \ldots & \textit{thread id} \\
\mu & ::= & \overline{l \mapsto [c\,.\,\overline{f \mapsto_\omega v}]} & \textit{store} \\
v & ::= & \texttt{null} \mid l & \textit{value} \\
\gamma & ::= & \overline{p \mapsto \langle b, \overline{l.m} \rangle} & \textit{event map} \\
b & ::= & \mathit{true} \mid \mathit{false} & \textit{boolean value} \\
\hbar & ::= & \overline{\langle id, \sigma \rangle} & \textit{trace} \\
\mathbb{E} & ::= & - \mid \mathbb{E}\,.m(\overline{e}) \mid v\,.m(\overline{v}\,\mathbb{E}\,\overline{e}) \mid \mathbb{E}\,.f \mid \mathbb{E}\,.f{=}e \mid v\,.f{=}\mathbb{E} \mid t\,x{=}\mathbb{E}; e & \textit{evaluation context} \\
 & & \mid \textbf{announce}\ p(\overline{v}\,\mathbb{E}\,\overline{e}) \mid \textbf{register}\ \mathbb{E}\,.m\ \textbf{with}\ p &
\end{array}
$$

Dynamic Typing: $\gamma, \mu, \Pi \vdash_{\mathcal{D}} e : t, \sigma$

$$
(\gamma\text{-}l)\ \frac{\mu(l) = [c.\overline{f \mapsto_\omega v}]}{\gamma, \mu, \Pi \vdash_{\mathcal{D}} l : c, \emptyset} \qquad (\gamma\text{-ANNOUNCE})\ \frac{\Pi \vdash \textbf{announce}\ p\ \ (\overline{e})\ : \texttt{void}, \sigma \qquad \gamma, \mu, \Pi \vdash_{\mathcal{D}} p : \langle b, \sigma' \rangle}{\gamma, \mu, \Pi \vdash_{\mathcal{D}} \textbf{announce}\ p\ \ (\overline{e})\ : \texttt{void}, \sigma \sqcup \sigma'}
$$

$$
(\gamma\text{-EVENT})\ \frac{\begin{array}{c}\gamma(p) = \langle b', \overline{l.m} \rangle \qquad b = \bigwedge \forall \sigma_i, \sigma_j \in \overline{\sigma}\ \text{s.t.}\ i \neq j\ \text{s.t.}\ \sigma_i \# \sigma_j \\ \forall l.m \in \overline{l.m}\ .\ \mathit{dispatch}(\mu, l, m) = m(\overline{t\,x})\{e\}\ \wedge\ \gamma, \mu, \overline{x : t} \vdash_{\mathcal{D}} [l/\texttt{this}]e : t', \sigma\end{array}}{\gamma, \mu, \Pi \vdash_{\mathcal{D}} p : \langle b, \sqcup \overline{\sigma} \rangle}
$$

For all other (γ-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of the judgment $\Pi \vdash e : t, \sigma$ in the latter rule should be substituted with $\gamma, \mu, \Pi \vdash_{\mathcal{D}} e : t, \sigma$ in the former.

Evaluation Relation: $\psi, \mu, \gamma, \hbar \hookrightarrow \psi', \mu', \gamma', \hbar'$

$$
(\mathit{cont})\quad \langle \mathbb{E}[e], id \rangle + \psi, \mu, \gamma, \hbar \ \hookrightarrow\ \langle \mathbb{E}[\textbf{yield}\ e'], id \rangle + \psi + \psi', \mu', \gamma', \hbar + \langle id, \sigma \rangle\ \textbf{if}\ \ e, id, \mu, \gamma \Rightarrow e', \psi', \mu', \gamma', \sigma
$$

Local Reduction: $e, _{\mathbb{C}} \Rightarrow e', \psi, \mu', \gamma', \sigma, \textbf{where}\ _{\mathbb{C}} = id, \mu, \gamma$

$$
\begin{array}{lll}
(\mathit{reg}) & \textbf{register}\ l.m\ \textbf{with}\ p, _{\mathbb{C}} \Rightarrow l, \emptyset, \mu, \gamma', \texttt{reg}\ p\ \sigma & \textbf{if}\ \mu(l) = [c.\overline{f \mapsto_\omega v}]\ \wedge\ \mathit{find}(c, m) = (\ldots, \sigma) \\
 & & \wedge\ \gamma'' = \gamma \oplus \{p \mapsto \langle b, \overline{l.m} + l.m \rangle\} \\
 & & \wedge\ \gamma' = \{p' \mapsto \langle b', \overline{l'.m'} \rangle \mid \gamma''(p') = \langle b'', \overline{l'.m'} \rangle\ \wedge\ \gamma'', \mu, \emptyset \vdash_{\mathcal{D}} p' : \langle b', \overline{\sigma} \rangle\} \\
(\mathit{ann}) & \textbf{announce}\ p\ (\overline{v}), _{\mathbb{C}} \Rightarrow e, \psi, \mu, \gamma, \texttt{ann}\ p & \textbf{if}\ \gamma(p) = \langle b, \overline{l.m} \rangle\ \wedge\ \psi = \overline{\langle e, id \rangle}\ \wedge\ e = \overrightarrow{\mathit{join}}(\overline{id}) \\
 & & \wedge \forall l_i.m_i \in \overline{l.m}\ id_i = id.\mathit{fresh}() \wedge e_i = \begin{cases} \mathit{dyn}(\mu, l_i, m_i, \overline{v}) & \textbf{if}\ b \\ \overrightarrow{\mathit{join}}(\overline{id_{i-1}}); \mathit{dyn}(\mu, l_i, m_i, \overline{v}) & \textbf{if}\ !b \end{cases} \\
(\mathit{join}) & \overrightarrow{\mathit{join}}(\overline{id}), _{\mathbb{C}} \Rightarrow e, \emptyset, \mu, \gamma, \texttt{join} & \textbf{if}\ \nexists id_i \in \overline{id}\ \text{s.t.}\ \langle e_i, id_i \rangle \in \psi\ \wedge\ e = \texttt{null}
\end{array}
$$

$$
\begin{array}{lrll}
(\mathit{call}) & l.m(\overline{v}), _{\mathbb{C}} \Rightarrow e, \emptyset, \mu, \gamma, \emptyset & \textbf{if}\ \mathit{dyn}(\mu, l, m, \overline{v}) = e \\
(\mathit{def}) & c\,x = v; e, _{\mathbb{C}} \Rightarrow e', \emptyset, \mu, \gamma, \emptyset & \textbf{if}\ e' = [v/x]e \\
(\mathit{new}) & \textbf{new}\ c(), _{\mathbb{C}} \Rightarrow l, \emptyset, \mu', \gamma, \emptyset & \textbf{if}\ l \notin \mathit{dom}(\mu)\ \wedge\ \mathit{flds}(c) = \overline{f \mapsto_\omega t}\ \wedge\ \mu' = \mu \oplus \{l \mapsto [c.\overline{f \mapsto_\omega \texttt{null}}]\} \\
(\mathit{set}) & l.f = v, _{\mathbb{C}} \Rightarrow v, \emptyset, \mu', \gamma, \textbf{wr}\ \omega & \textbf{if}\ \mu' = \mu \oplus (l \mapsto [c.\overline{f \mapsto_\omega v} \oplus (f \mapsto_\omega v)]) \\
(\mathit{get}) & l.f, _{\mathbb{C}} \Rightarrow v, \emptyset, \mu, \gamma, \textbf{rd}\ \omega & \textbf{if}\ \mu(l) = [c.\overline{f \mapsto_\omega v}]
\end{array}
$$

Cooperative Handling: $\psi, \mu, \gamma, \hbar \hookrightarrow \psi', \mu', \gamma', \hbar'$

$$
\begin{array}{lrcl}
(\mathit{yield}) & \langle \langle \mathbb{E}[\textbf{yield}\ e], id \rangle + \psi, \mu, \gamma, \hbar \rangle & \hookrightarrow & \langle \mathit{active}(\psi + \langle \mathbb{E}[e], id \rangle), \mu, \gamma, \hbar \rangle \\
(\mathit{end}) & \langle \langle v, id \rangle + \psi, \mu, \gamma, \hbar \rangle & \hookrightarrow & \langle \mathit{active}(\psi), \mu, \gamma, \hbar \rangle
\end{array}
$$

Figure 8. Operational semantics. Auxiliary functions are defined in Figure 9.

Dynamic Dispatch, $dispatch(\mu, l, m) = m(\overline{t\,x})\{e\}$:

$$dispatch(\mu, l, m) = m(\overline{t\,x})\{e\} \quad \textbf{if } \mu(l) = [c.\overline{f \mapsto_\omega v}] \wedge find(c, m) = (c', t, m(\overline{t\,x})\{e\}, \sigma)$$

$$dyn(\mu, l, m, \overline{v}) = [l/\texttt{this}, \overline{v/x}]e \quad \textbf{if } dispatch(\mu, l, m) = m(\overline{t\,x})\{e\}$$

Cooperative Handlers Management, $active(\psi) = \psi$:

$$active(\langle e, id\rangle + \psi) = \begin{cases} \langle e, id\rangle + \psi & \textbf{if } e \neq \overrightarrow{join}(\overline{id}) \\ & \vee\, \nexists id' \in \overline{id} \text{ s.t. } \langle e', id'\rangle \in \psi \\ active(\psi + \langle e, \tau\rangle) & \textbf{otherwise} \end{cases}$$

Figure 9. Auxiliary functions for the semantics.

ative semantics, where the only interference points appear after the **announce** expression.

The (YIELD) puts the current handler to the end of the queue ψ and starts the next active task from this queue. Finding an active task is done by the function $active$ (Figure 9). It returns the top most task in ψ that can be run. A task is ready to run if it is not waiting on other tasks, *i.e.*, not a $\overrightarrow{join}$ expression, or all the tasks it is waiting on are done.

The (END) rule says that the current running task is done (it evaluates to a single value v), thus it will be removed from the queue and the next active task will be scheduled.

5. Meta-Theories

We now show the key properties of ATE. The properties include the standard type soundness (§5.3), liveness (§5.2), sequential semantics (§5.4) and sparse interference points (§5.5). In previous works [6, 37], the exact set of concurrent tasks that will be spawned are known statically. A technical challenge for proving the soundness of our work is that concurrent tasks spawned as a result of an event announcement are unknown statically due to dynamic registration. The detailed proofs can be found in our report [25].

5.1 Preliminary Definitions

Before we proceed, we first give some simple definitions that will be used for the rest of the section.

DEFINITION 5.1. *[Redex configuration] We say Σ is a* redex configuration *of program $\overline{decl}$ e, written $e \trianglerighteq \Sigma$, iff $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \overset{*}{\hookrightarrow} \Sigma$. We say Σ is a proper redex configuration, written $\trianglerighteq\Sigma$, if $\exists e$ such that $\vdash \overline{decl}\ e : t, \sigma$ and $e \trianglerighteq \Sigma$.*

DEFINITION 5.2. *[Well-typed queue] A queue ψ is* well-typed *in μ and γ, written $\gamma, \mu \vdash \psi$, if and only if $\forall \langle e, id\rangle \in \psi, \gamma, \mu, \emptyset \vdash_D e : t, \sigma$ for some t and σ.*

DEFINITION 5.3. *[Well-typed event map] A handler $l.m$ is* well-typed *in μ for event p, written as $l.m \simeq (\mu, p)$, if $(m\ \overline{(t'\,x)}\ \{e\}) = dispatch(\mu, l, m)$, $\textbf{event}\ p\ \{\overline{t\,x}\} \in CT$, $(\forall\ t_i\ x_i \in \overline{t\,x},\ t'_i \preccurlyeq t_i)$. An event map γ is* well-typed *in μ, written as $\mu \vdash \gamma$, if $\forall p \in dom(\gamma)$ s.t. $(\gamma(p) = \langle b, \overline{l.m}\rangle) \Rightarrow (\forall l.m \in \overline{l.m}$ s.t. $l.m \simeq (\mu, p))$.*

DEFINITION 5.4. *[Local reduction] Let two configurations $\Sigma = \langle\langle e, id\rangle + \psi, \mu, \gamma, \hbar\rangle$ and $\Sigma' = \langle\langle e', id\rangle + \psi', \mu', \gamma', \hbar'\rangle$. A reduction $\Sigma \hookrightarrow^* \Sigma'$, is called a task* local reduction*, denoted as $\Sigma \rightarrowtail \Sigma'$, if $\nexists e''$ s.t. $\Sigma \hookrightarrow^* \langle\langle e'', id\rangle + \psi'', \mu'', \gamma'', \hbar''\rangle \hookrightarrow^* \Sigma'$.*

DEFINITION 5.5. *[Well-typed configuration] A configuration $\Sigma = \langle\psi, \mu, \gamma, \hbar\rangle$ is* well-typed*, written as $\vdash \Sigma$, if $\gamma, \mu \vdash \psi$ and $\mu \vdash \gamma$.*

5.2 Livelock Freedom

In the semantics, ATE lets some tasks be inactive, *i.e.*, wait for conflicting tasks to maintain concurrency safety. We prove that at any configuration, there exists a task that is active and thus can make progress. Intuitively, each task can only wait for its predecessor handlers, *i.e.*, conflicting handlers that registered earlier and its handlers if it announces an event. Such waiting relationship forms a tree, not a circle (circular wait). Therefore, ATE is livelock free.

DEFINITION 5.6. *[Blocked configurations] Let a configuration $\Sigma = \langle\psi, \mu, \gamma, \hbar\rangle$. The task $\langle e, id\rangle$ in Σ blocks, written $\uparrow \langle e, id\rangle$, if $e == \mathbb{E}[\overrightarrow{join}(\overline{id})]$ and $\exists \langle e', id'\rangle \in \psi$ s.t. $id' \in \overline{id}$. Otherwise, $\langle e, id\rangle$ is active, written $\downarrow\langle e, id\rangle$. A configuration Σ blocks, written $\uparrow \Sigma$, if $\forall \langle e, id\rangle \in \psi, \uparrow \langle e, id\rangle$, otherwise, Σ can make progress, written $\downarrow \Sigma$.*

THEOREM 5.7. [Liveness] *If $\trianglerighteq\Sigma$, then $\downarrow \Sigma$.*

Proof: The proof is by induction on the number of reduction steps (Figure 8) applied.

5.3 Type Soundness

In this section, we prove the standard type soundness. First we prove that with more items in the store μ and event map γ, the typing of the same expression remain unchanged.

DEFINITION 5.8. *[Store enlargement] Let μ and μ' be two stores. We write $\mu \lessdot \mu'$, if:*

1. *$dom(\mu) \subseteq dom(\mu')$;*
2. *$\forall l \in dom(\mu)$, if $\mu(l) = [c.\overline{f \mapsto_\omega v}]$, then $\mu'(l) = [c.\overline{f \mapsto_\omega v'}]$.*

LEMMA 5.9. [Store extension] *If $\gamma, \mu, \Pi \vdash_D e : t, \sigma$ and $\mu \lessdot \mu'$, then $\gamma, \mu', \Pi \vdash_D e : t, \sigma$.*

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$.

DEFINITION 5.10. *[Event map enlargement] Let p be an event type, γ and γ' be two event maps. We write $\gamma \lessdot_{\langle p, l.m\rangle} \gamma'$, if all the following hold:*

1. *$dom(\gamma) = dom(\gamma')$;*
2. *$\forall p' \in dom(\gamma)$, if $\gamma(p') = \langle b, \overline{l.m}\rangle$, then $\gamma'(p') = \langle b', \overline{l.m}\rangle$;*
3. *if $\gamma(p) = \langle b, \overline{l.m}\rangle$, then $\gamma(p) = \langle b', \overline{l.m} + l.m\rangle$.*

LEMMA 5.11. [Event map extension I] *If $\gamma, \mu, \Pi \vdash_D e : t, \sigma$, $\gamma \lessdot_{\langle p, l.m\rangle} \gamma'$, and $\textbf{ann}\ p \notin \sigma$ then $\gamma', \mu, \Pi \vdash_D e : t, \sigma$.*

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$.

Evaluation Relation: $\psi, \mu, \gamma, \hbar \hookrightarrow_S \psi', \mu', \gamma', \hbar'$

$$(cont) \quad \langle \mathbb{E}[e], id\rangle + \psi, \mu, \gamma, \hbar \quad \hookrightarrow_S \quad \langle \mathbb{E}[e'], id\rangle + \psi + \psi', \mu', \gamma', \hbar + \langle id, \sigma\rangle \ \textbf{if} \quad e, id, \mu, \gamma \Rightarrow_S e', \psi', \mu', \gamma', \sigma$$

Sequential Reduction: $e_{\mathbb{C}} \Rightarrow_S e', \psi, \mu', \gamma', \sigma$ **where** ${}_{\mathbb{C}} = id, \mu, \gamma$

$$(ann_S) \quad \textbf{announce}\ p\ (\overline{v})_{\mathbb{C}} \quad \Rightarrow_S \quad \overline{e}, \emptyset, \mu, \gamma, \textbf{ann}\ p \qquad \textbf{if}\ \gamma(p) = \langle b, \overline{l.m}\rangle \quad \wedge \quad \forall l_i.m_i \in \overline{l.m}.\ dyn(\mu, l_i, m_i, \overline{v}) = e_i$$

For all other ($*_S$) rules, each is isomorphic to its counterpart (*) rule, except that every occurrence of the judgment $e_{\mathbb{C}} \Rightarrow e', \psi, \mu', \gamma', \sigma$ in the latter rule should be substituted with $e_{\mathbb{C}} \Rightarrow_S e', \psi, \mu', \gamma', \sigma$ in the former.

Figure 10. Sequential semantics.

LEMMA 5.12. [Event map extension II] *If* $\gamma, \mu, \Pi \vdash_D e : t, \sigma$, $\gamma \lessdot_{\langle p,l.m\rangle} \gamma'$, $\mu(l) = [c.\overline{f \mapsto_\omega v}]$, $find(c, m) = (\ldots, \sigma')$, *and* **ann** $p \in \sigma$ *then* $\gamma', \mu, \Pi \vdash_D e : t, post(\sigma \cup \sigma')$.

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$.

Our soundness proof is constructed through standard subject reduction and progress:

LEMMA 5.13. [Preservation] *Let* $\Sigma = \langle\langle e, id\rangle + \psi, \mu, \gamma, \hbar\rangle$. *If* $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, $\Sigma \rightarrowtail \langle\langle e', id\rangle + \psi', \mu', \gamma'\rangle$, *then there is some* t' *and* σ' *such that* $\gamma', \mu', \emptyset \vdash e' : t', \sigma' \wedge t' \preccurlyeq t$.

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$, Lemma 5.9, 5.12, and 5.12.

LEMMA 5.14. [Progress] *Let* $\Sigma = \langle\langle e, id\rangle + \psi, \mu, \gamma, \hbar\rangle$. *If* $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, *then either* e *is a value* v, *or* $\Sigma \rightarrowtail \langle\langle e', id\rangle + \psi', \mu', \gamma'\rangle$.

Proof: By cases on the reduction step applied.

THEOREM 5.15. [Type Soundness] *Given an expression* e, $\emptyset, \emptyset, \emptyset \vdash_D e : t, \sigma$, *then either the evaluation of* e *diverges, or there exists some* μ, v, γ *and* $\hbar$ *such that* $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \hookrightarrow^* \langle\langle v, 0\rangle, \mu, \gamma, \hbar\rangle$.

Proof: By Lemma 5.14 and 5.13.

5.4 Sequential Semantics

In this section, we will prove that the execution of an ATE program (which runs nonconflicting handlers concurrently in §4) is *behaviorally equivalent* to its sequential counterparts, where every **announce** expression will execute the handlers one by one serially. First, let us define relation $\hbar \propto \sigma$, which says a trace $\hbar$ *realizes* a static effect σ:

DEFINITION 5.16. *[Static effect contains dynamic effect]* $\hbar \propto \sigma$ *holds iff* $\forall \langle id, \sigma'\rangle \in \hbar\ .\ \sigma' \subseteq \sigma$.

Next, we state and prove that every pair of the handlers in the queue ψ do not conflict:

DEFINITION 5.17. *[Noninterfering tasks] Two task* $\langle e, id\rangle$ *and* $\langle e', id'\rangle$ *are* noninterfering *in* μ, γ, *written* $\gamma, \mu \trianglerighteq \langle e, id\rangle \# \langle e', id'\rangle$, *if:*

1. $e = \overrightarrow{join}(\overline{id}) \ \wedge \ id' \in \overline{id}$; *or* $e' = \overrightarrow{join}(\overline{id}) \ \wedge \ id \in \overline{id}$;
2. *or* $\gamma, \mu, \emptyset \vdash_D e : \sigma, t$, $\gamma, \mu, \emptyset \vdash_D e' : \sigma', t'$ *and* $\sigma \# \sigma'$.

DEFINITION 5.18. *[Noninterfering queue] A queue* ψ *is* noninterfering *in* μ *and* γ, *written* $\gamma, \mu \trianglerighteq \psi$, *if* $\forall \langle e_i, id_i\rangle, \langle e_j, id_j\rangle \in \psi$ *s.t.* $i \neq j, \gamma, \mu \trianglerighteq \langle e_i, id_i\rangle \# \langle e_j, id_j\rangle$.

LEMMA 5.19. [Noninterfering preservation] *Let* $\Sigma = \langle\psi, \mu, \gamma, \hbar\rangle$, *and* $\trianglerighteq \Sigma$. *If* $\gamma, \mu \trianglerighteq \psi$, $\Sigma \hookrightarrow \Sigma'$ *where* $\Sigma' = \langle\psi', \mu', \gamma', \hbar'\rangle$, *then* $\gamma', \mu' \trianglerighteq \psi'$.

Proof: By cases on the reduction step and Definition 5.18.

Next, we prove that the trace a handler leaves realizes its effects given by the dynamic typing:

LEMMA 5.20. [Effect subsumption] *Let* $\Sigma = \langle\langle e, id\rangle, \mu, \gamma, \hbar\rangle$, *and* $\trianglerighteq \Sigma$. *If* $\gamma, \mu \trianglerighteq \psi$, $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, $\Sigma \rightarrowtail \Sigma'$ *where* $\Sigma' = \langle\langle e', id\rangle + \psi', \mu', \gamma', \hbar + \hbar'\rangle$, *then*

(a) $\gamma', \mu', \emptyset \vdash_D e' : t', \sigma' \wedge (t' \preccurlyeq t) \wedge (\sigma' \subseteq \sigma)$;
(b) $\hbar' \propto \sigma$.

Proof: By cases on the reduction step applied.

Prefix, $pref(id, id') = b$:

$$pref(id, id') = \begin{cases} true & \textbf{if}\ (id == id') \vee (id == id'.id'' \text{ for some } id'') \\ false & \textbf{otherwise} \end{cases}$$

Trace Projection, $\pi(id, \hbar) = \hbar$:

$$\pi(id, \hbar) = \begin{cases} \emptyset & \textbf{if}\ \hbar = \emptyset \\ \langle id', \sigma\rangle + \pi(id, \hbar') & \textbf{if}\ \hbar = \langle id', \sigma\rangle + \hbar' \ \wedge \ pref(id, id') \\ \pi(id, \hbar') & \textbf{otherwise} \end{cases}$$

Figure 12. Trace projection.

LEMMA 5.21. [Effect preservation] *If* $\Sigma = \langle\langle e, id\rangle, \mu, \gamma, \hbar\rangle$, *and* $\trianglerighteq \Sigma$. *If* $\gamma, \mu \trianglerighteq \psi$, $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, $e \neq \mathbb{E}[\overrightarrow{join}(\overline{id})]$, $\Sigma \hookrightarrow^* \langle\langle v, id\rangle, \mu', \gamma', \hbar'\rangle$ *then* $\pi(id, \hbar' - \hbar) \propto \sigma$.

Proof: By cases on the reduction step applied.

With the above, we can prove that any ATE program is race free. There remains a gap between this property and why one *intuitively* believes that ATE is sequentially consistent (SC). To rigorously define the more intuitive notion of SC, let us first introduce the sequential semantics (handlers run sequentially) of ATE, shown in Figure 10.

Evaluation Relation: $\psi, \mu, \gamma, \hbar \hookrightarrow_{Y} \psi', \mu', \gamma', \hbar'$

$$(cont)\ \langle \mathbb{E}[e], id\rangle + \psi, \mu, \gamma, \hbar \hookrightarrow_{Y} \langle \mathbb{E}[e'], id\rangle + \psi + \psi', \mu', \gamma', \hbar + \langle id, \sigma\rangle \ \textbf{if}\ e, id, \mu, \gamma \Rightarrow_{Y} e', \psi', \mu', \gamma', \sigma$$

Cooperative Reduction: $e_{\mathbb{C}} \Rightarrow_{Y} e', \psi, \mu', \gamma', \sigma$ **where** $_{\mathbb{C}} = id, \mu, \gamma$

$$(ann_{Y})\quad \textbf{announce}\ p\ (\overline{v})_{\mathbb{C}} \Rightarrow_{Y} \textbf{yield}\ e, \psi, \mu, \gamma, \texttt{ann}\ p \quad \textbf{if}\ \textbf{announce}\ p\ (\overline{v})_{\mathbb{C}} \Rightarrow e, \psi, \mu, \gamma, \texttt{ann}\ p$$

For all other ($*_{Y}$) rules, each is isomorphic to its counterpart (*) rule, except that every occurrence of the judgment $e_{\mathbb{C}} \Rightarrow e', \psi, \mu', \gamma', \sigma$ in the latter rule should be substituted with $e_{\mathbb{C}} \Rightarrow_{Y} e', \psi, \mu', \gamma', \sigma$ in the former.

Figure 11. Cooperative semantics.

We are ready to prove that an ATE program behaves the same as its sequential counterpart:

THEOREM 5.22. [Sequential Semantics] *Given an expression e, $\emptyset, \emptyset, \emptyset \vdash_{D} e : t, \sigma$, then either the evaluation of e diverges in both the sequential and the parallel semantics, or there exists some μ, v, γ, $\hbar$ and $\hbar'$ such that $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \hookrightarrow^{*} \langle\langle v, 0\rangle, \mu, \gamma, \hbar\rangle$ and $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \hookrightarrow^{*}_{S} \langle\langle v, 0\rangle, \mu, \gamma, \hbar'\rangle$.*

Proof: By Lemma 5.19, 5.20, and 5.21.

5.5 Modular Reasoning

In this section, we will prove that the execution of an ATE program (which has preemptive semantics, *i.e.*, yielding control to other active handlers at each step in §4) is *behaviorally equivalent* to its cooperative counterparts: a handler will only yield after announcing an event. The cooperative semantics is defined in Figure 11.

THEOREM 5.23. [Cooperative Semantics] *Given an expression e, $\emptyset, \emptyset, \emptyset \vdash_{D} e : t, \sigma$, then either the evaluation of e diverges in both the cooperative and the parallel semantics, or there exists some μ, v, γ, $\hbar$ and $\hbar'$ such that $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \hookrightarrow^{*} \langle\langle v, 0\rangle, \mu, \gamma, \hbar\rangle$ and $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \hookrightarrow^{*}_{Y} \langle\langle v, 0\rangle, \mu, \gamma, \hbar'\rangle$.*

Proof: As proven in §5.4, an ATE program is data race free. Therefore, reference access is both mover [39]. The **announce** expression, which forks concurrent handlers, is a left mover, and the $\overrightarrow{join}$ expression, which waits for its children handlers, is a right mover. Interference points only exist at left movers [39], *i.e.*, the **announce** expression.

6. Related Work

Asynchronous typed events are inspired from a large body of work on events, *e.g.*, [7, 9, 12, 13, 19, 21, 27, 28, 31, 35, 36]. This work goes beyond previous work which views events as a design decoupling mechanisms [12, 27, 31, 36] to leverage decoupling for safe concurrency.

There are plenty of works on using static effect systems to reason about safe concurrency. Earlier work includes Lucassen [26], and Talpin et al. [37], and more recent examples such as Task Types [20] and Bocchino et al. [6]. Compared with these works, our system uses the effects dynamically. Effects of the handlers are computed statically by the type system, and the semantics use these effects to compute a safe order for handler invocation at runtime.

There are several works on using effects dynamically, including Intensional Effect Polymorphism [23], TWEJava [16], and Legion [38]. In these works, effects do not change at runtime. In ATE, however, effects could change due to dynamic event registration. ATE introduces a novel type-and-effect system to reason about mutable handler queue, which could be challenging for the above systems.

Compared with software transactional memory (STM) and other related ideas [5] and effect monitoring systems [4], our system computes the effects of the handlers by the type system. Concurrency decisions are guided by the effects of handlers at runtime to gain precision. ATE detects potential conflicts before executing the handlers, while STM executes threads speculatively, and detects conflicts afterwards. In case of conflicts, STM rolls back all the changes.

There is a large body of work on the message-passing, and the publish/subscribe paradigms in distributed systems [22, 28, 30, 34]. These works either require programmers to manually account for data races, or assume disjoint address space between concurrent processes [28, 34]. ATE tackles concurrency problems in shared-memory paradigm. It eases the burden on the programmer by allowing modular reasoning and by providing implicit safe concurrency.

7. Conclusion

In this work, we pursue the goal of unifying modular reasoning and concurrency in program design. We have developed the notion of *asynchronous, typed events* that are helpful for programs where modules are decoupled using implicit-invocation design style [27, 28, 36], and where handlers can register dynamically. Event announcements provide implicit concurrency. Registration-time specialization provides precise effects analyses, which improves safe concurrency for II programs. Dynamic typing takes into account the handlers registered to reason about the mutable handler queue. ATE facilitates modular reasoning about concurrency safety.

Acknowledgments

Tyler Sondag and Sean L. Mooney helped with an earlier version of this work. This work was supported in part by NSF grants CCF-08-46059, CCF-11-17937 and CCF-14-23370. Comments and suggestions from Mehdi Bagherzadeh, Robert Dyer, Steven M. Kautz, Gary T. Leavens, Youssef Hanna, Adam Zimmerman, Cody Hanika, John L. Singleton, Rex D. Fernando, Zhaotong Zhang and the anonymous reviewers of Modularity '16 were helpful.

References

[1] Abadi, M., Plotkin, G.: A model of cooperative threads. In: POPL '09

[2] Bagherzadeh, M., Dyer, R., Fernando, R.D., Sánchez, J., Rajan, H.: Modular reasoning in the presence of event subtyping. In: MODULARITY '15

[3] Bagherzadeh, M., Rajan, H.: Panini: A concurrent programming model for solving pervasive and oblivious interference. In: MODULARITY '15

[4] Bañados, F., Garcia, R., Tanter, É.: A theory of gradual effect systems. In: ICFP '14

[5] Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for C/C++. In: OOPSLA '09

[6] Bocchino, Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In: OOPSLA '99

[7] Bodden, E., Tanter, E., Inostroza, M.: Join point interfaces for safe and flexible decoupling of aspects. ACM Trans. Softw. Eng. Methodol. 23(1) (2014)

[8] Clifton, C., Leavens, G.T.: MiniMAO$_1$: Investigating the semantics of proceed. Sci. Comput. Program 63(3) (2006)

[9] Cunningham, R., Kohler, E.: Making events less slippery with eel. In: HOTOS '05

[10] Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: ECOOP '06

[11] Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: ICSE '13

[12] Eugster, P., Jayaram, K.R.: EventJava: An extension of Java for event correlation. In: ECOOP '09

[13] Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: PEPM '07

[14] Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: PLDI '09

[15] Greenhouse, A., Boyland, J.: An object-oriented effects system. In: ECOOP '99

[16] Heumann, S.T., Adve, V.S., Wang, S.: The tasks with effects model for safe concurrency. In: PPoPP '13

[17] Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. 39(12)

[18] Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA '99

[19] Krohn, M., Kohler, E., Kaashoek, M.F.: Events can make sense. In: USENIX (2007)

[20] Kulkarni, A., Liu, Y.D., Smith, S.F.: Task types for pervasive atomicity. In: OOPSLA '10

[21] Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: PLDI '07

[22] Long, Y., Bagherzadeh, M., Lin, E., Upadhyaya, G., Rajan, H.: On ordering problems in message passing software. In: MODULARITY '16

[23] Long, Y., Liu, Y.D., Rajan, H.: Intensional effect polymorphism. In: ECOOP '15

[24] Long, Y., Mooney, S.L., Sondag, T., Rajan, H.: Implicit invocation meets safe, implicit concurrency. In: GPCE '10

[25] Long, Y., Rajan, H.: A type-and-effect system for asynchronous, typed events. Tech. Rep. 09-28b, Iowa State U., Dept. of Computer Sc. (2015)

[26] Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL '88

[27] Notkin, D., Garlan, D., Griswold, W.G., Sullivan, K.J.: Adding implicit invocation to languages: Three approaches. In: JSSST '93

[28] Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The information bus: An architecture for extensible distributed systems. In: SOSP '93

[29] Pierce, B.C.: Types and Programming Languages. MIT Press (2002)

[30] Rajan, H.: Capsule-oriented programming. In: ICSE'15

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

[32] Raychev, V., Vechev, M., Sridharan, M.: Effective race detection for event-driven programs. In: OOPSLA '13

[33] Safi, G., Shahbazian, A., Halfond, W.G.J., Medvidovic, N.: Detecting event anomalies in event-based systems. In: FSE '15

[34] Schmidt, D.C.: Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching. Pattern languages of program design pp. 529–545 (1995)

[35] Steimann, F., Pawlitzki, T., Apel, S., Kästner, C.: Types and modularity for implicit invocation with implicit announcement. ACM Trans. Softw. Eng. Methodol. 20(1) (2010)

[36] Sullivan, K.J., Notkin, D.: econciling environment integration and component independence. SIGSOFT Software Engineering Notes 15(6), 22–33 (December 1990)

[37] Talpin, J.P., Jouvelot, P.: The type and effect discipline. Inf. Comput. 111(2) (1994)

[38] Treichler, S., Bauer, M., Aiken, A.: Language support for dynamic, hierarchical data partitioning. In: OOPSLA '13

[39] Yi, J., Flanagan, C.: Effects for cooperable and serializable threads. In: TLDI '10