A Type-and-Effect System for Shared Memory, Concurrent Implicit Invocation Systems

Yuheng Long, Tyler Sondag, and Hridesh Rajan

TR #10-09a
Initial Submission: December 15, 2010.
Revised: June 1, 2011.

Keywords: Hybrid type-and-effect System, implicit-invocation languages, aspect-oriented programming languages, event types, event expressions, concurrent languages.
CR Categories:
D.2.10 [*Software Engineering*] Design
D.1.5 [*Programming Techniques*] Object-Oriented Programming
D.2.2 [*Design Tools and Techniques*] Modules and interfaces,Object-oriented design methods
D.2.3 [*Coding Tools and Techniques*] Object-Oriented Programming
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures

Copyright (c) 2011, Yuheng Long, Tyler Sondag, and Hridesh Rajan.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

A Hybrid Type-and-Effect System for Shared Memory, Concurrent Implicit Invocation Systems

Yuheng Long Tyler Sondag Hridesh Rajan

Dept. of Computer Science, Iowa State University

{csgzlong,sondag,hridesh}@iastate.edu

Abstract

The notion of events in *distributed publish-subscribe* systems implies safe concurrency. However, that implication does not hold in object-oriented (OO) programs that utilize events for modularity. This is because unlike the distributed setting, where publisher and subscriber do not share state and only communicate via messages, additional communication between publisher and subscriber, e.g. via call-back or shared state, is common in OO programs that use events.

Static type-and-effect systems can help expose potential concurrency, however, they are too conservative to handle an event-based idiom that involves zero or more dynamic dispatches on receiver objects in a dynamically changing list. To solve these problems, we present a hybrid (static/-dynamic) type-and-effect system for exposing concurrency in event-based OO programs. This type-and-effect system provides deadlock and data race freedom in such usage of the event idiom. We have implemented this type-and-effect system as an extension of Java's type system and it shows considerable speedup over the sequential version of several applications (up to 15.7x) at a negligible overhead.

1. Introduction

Use of events is thought to naturally result in concurrency. This is indeed true for *publish/subscribe*-based distributed systems [18, 42, 52] where events help decouple the execution of components, thereby exposing potential concurrency in system design [52]. In shared memory programs, however, use of events as a decoupling mechanism, e.g. via the implicit-invocation design style [27] that is embodied in such design patterns as observer and mediator[26], does not necessarily expose concurrency in program design safely.

[Copyright notice will appear here once 'preprint' option is removed.]

In shared memory programs, sharing and callbacks between publishers and subscribers can introduce concurrency-related hazards. This work presents a hybrid type-and-effect system that helps realize the benefits of the event abstraction towards exposing safe concurrency in programs.

1.1 FindBugs: A Running Example

To illustrate the use of implicit invocation towards exposing potential concurrency, consider the code snippets in Figure 1 that are taken from the code of the FindBugs tool [31]. FindBugs is a static analysis tool to detect bugs. It is widely used in practice and provides plugins for many IDEs [1].

Figure 1 shows one of the driver classes for this application (`FindBugs` on line 1). This class orchestrates the analysis of projects and collects results. It holds an array of bug detectors (line 4), which are then used to analyze any input Java application (line 6). For future evolution, e.g. add/remove bug detectors, it is desirable that the class `FindBugs` remains independent of bug detectors. This decoupling is achieved using the observer pattern [26] with the class `FindBugs` as a subject and `Detector`s as handlers. The function `analyze` uses a `list` to store active bug detectors (line 4) and invokes all bug detectors in `list` without naming their concrete classes (lines 5-6). The concrete bug detectors implement the interface `Detector` (not shown here) that has a handler method `check`. Concrete bug detectors (e.g. `MutableLock` and `FindBadCast`) implement different types of bug detection.

Once a bug is detected, reporting it (using interface `Reporter`'s method `reportBug`) is desirable. This reporting is triggered by the bug detectors. The type of reporting depends on options that can be set dynamically. It would be sensible to keep the bug detector modules separate from the `Reporter` modules. By keeping these modules separate, both can be reused. This design goal is also achieved using the observer pattern. In this scenario, `Detector`s are subjects and the `Reporter`s are handlers.

Similar to the subject `FindBugs`, the class `MutableLock` implements the functionality to hold a handler (instance of `Reporter`, line 18) and will notify this handler when a bug is found (line 24).

```
1 class FindBugs {
2   void analyze() {
3     Class c = /*...*/;
4     Detector[] list = /*...*/;
5     for (Detector d : list){
6       d.check(c);
7     }
8   }
9 }
10 class XDocsReport implements Reporter {
11   Element root;
12   void reportBug(Bug bi){
13     root.addElement(/*...*/);
14   }
15 }
```

```
16 class MutableLock
17                 implements Detector {
18   Reporter r;
19   MutableLock(Reporter r){
20     this.r = r;
21   }
22   void check(Class c){
23     if(/*...*/) {
24       r.reportBug(/*...*/);
25     }
26   }
27 }
```

```
28 class FindBadCast
29                 implements Detector {
30   Reporter r;
31   FindBadCast(Reporter r){
32     this.r = r;
33   }
34   void check(Class c){
35     if(/*...*/) {
36       r.reportBug(/*...*/);
37     }
38   }
39 }
```

Figure 1. Snippets adapted from FindBugs [31] that uses implicit invocation to decouple bug detection and reporting classes.

1.2 The Problems and their Importance

To enhance the scalability of FindBugs, it may be desirable to expose concurrency in its design. To analyze applications, users generally activate several concrete detectors from a variety of choices. Each detector analyzes the entire input application, and thus is compute-intensive. Therefore, running multiple detectors concurrently can enhance scalability.

```
1 class FindBugs {
2   void analyze() {
3     final Class c = /*...*/;
4     Detector[] list = /*...*/;
5     int size = list.length;
6     Thread[] t = new Thread[size];
7     for(int i = 0; i < size; i++){
8       final Detector d = list[i];
9       t[i] = new Thread(new Runnable(){
10        public void run(){ d.check(c);}
11      });
12      t[i].start();
13    }
14    /* joins the threads in t */
15  }
16 }
```

Figure 2. Concurrent implementation of the method `analyze` in `FindBugs` using Java Threads.

Figure 2 shows a common idiom for exposing this concurrency between detectors using Java threads. It creates one thread for each detector, spawns these threads, and joins them after completion. This idiom is effective, but may result in concurrency-related hazards [29].

For example, in Figure 1, multiple bug detectors may report bugs concurrently (the reporter is shared by all detectors). Further, each reporting may modify a common structure as is the case in reporter `XDocsReport` for method `addElement` which is concurrency-unsafe and does not commute [47]. Thus, concurrently invoking this method on the shared `Reporter` object will result in concurrency-unsafe hazards [29]. Therefore, we may have a race condition and non-deterministic behavior.

Previous work has proposed type-and-effect systems [28, 54] to solve these problems [11, 12, 45]. The basic idea behind a type-and-effect system is to statically check read and write effects such that these checks are a conservative approximation of effects that may happen at runtime.

Conservative Approximations. Even if only certain rare control flow paths are concurrency-unsafe, a static type-and-effect system for validating concurrent programs will declare such a program concurrency-unsafe [30, 37, 45].

Overly conservative approximation in a type-and-effect system may be caused by two features: data structures with dynamically varying number of elements (e.g. `List`) and dynamic dispatch [50, 55]. The effects of an operation on such a data structure must be taken as the upper bound of effects of this operation on any possible state of the structure. The effects of a dynamically dispatched method call must be taken as the upper bound of effects produced by all overriding implementations of the called method.

Unfortunately, these features are both used in a typical observer implementation as seen in Figure 1 (lines 4, 6, 24, and 36). For example, the method `analyze`, on line 2 holds an array (dynamically varying number of elements) of `Detectors`. The actual type of these `Detectors` could be any concrete subtype of `Detector` (dynamic dispatch).

To illustrate the effect of conservative approximations on potential concurrency, let us consider the concurrent version of the `analyze` method in Figure 2. Informally, the method `analyze` is concurrency-safe if the effects of the `check` methods in the concrete `Detector` classes are disjoint.

Since we can not statically know which concrete elements (subtypes of `Detector`) the `list` will hold, we must analyze with respect to all possible subtypes of `Detector`. One such subtype is `MutableLock`. The effects of the `check` method in `MutableLock` include the effects of the method `reportBug` (since it may be invoked on some possible execution path). Since we can not know statically which concrete subtype of `Reporter` will be used at runtime, the effect set of `check` must include the effect set of `XDocsReport`. Therefore, the effect of the `check` method for the `MutableLock` class will contain the effects of `addElement` which is concurrency-unsafe. This effect also applies to the class `FindBadCast` and other concrete subtypes of `Detector`. Thus, a sound static type-and-effect system will conclude that concurrently executing the detectors could lead to race conditions and reject this implementation as potentially concurrency-unsafe.

However, there are many scenarios in which FindBugs can reap safe concurrency benefits. For example, instead of using `XDocsReport`, other reporters could be used which are concurrency-safe leading to safe concurrency. Thus, a carefully designed type-and-effect system with runtime information could enjoy concurrency benefits which are sacrificed by a static type-and-effect system.

Runtime Checks. There are several systems that use just dynamic mechanisms to expose potential concurrency in program design. However, in general the runtime overhead of instrumentation essential to determine if a concurrency-unsafe control flow path is about to run is often prohibitive for production runs [21, 49]. Finally, many of these techniques require special hardware [35, 43]. Thus, programs that require sound safety guarantees must choose between *lack of potential concurrency* and *prohibitive overhead*.

1.3 Contributions

The main novelty of this work stems from our insight that, even though in general detecting whether a concurrency-unsafe control flow is about to run may have prohibitive costs, for implicit invocation mechanisms it can be done at an acceptable cost. This is based on two observations.

1. Handler registrations are infrequent compared to event announcements. For example, in FindBugs, a reporter is registered only once (line 10) at the very beginning of the program. However, reporters are used frequently, i.e. whenever a bug is found (line 24 and line 36).
2. The exact set of tasks that will be run when an event is signaled and their potential conflicts can be computed during handler registration. For example, when the bug reporter registers on line 10, the effect set for the method `check` of class `MutableLock` may be computed.

Because of these observations, we hypothesize that exposing safe concurrency for implicit invocation mechanisms can be done accurately and at an acceptable cost.

Building on this insight, we formally define a hybrid type-and-effect system for programs written in the II design style. This system introduces two new effects, namely *announce* and *register*, expressed as **ann** and **reg**. Similar to other static analyses, this hybrid system computes effect summaries for every method. Unlike previous work, our system uses these two newly introduced effects dynamically. An **ann** effect serves as a placeholder for the concrete effects of zero or more registered handlers and is made concrete during handler registration. Since the exact set of handlers is known during registration, the placeholder effect **ann** is taken as the union of the effects of registered handlers.

For example, on line 6 in Figure 1, the type-and-effect system determines the effect of the method `check` of class `MutableLock` to be {**ann** }. For now, the type-and-effect system assumes an announce effect does not conflict with another announce effect. Thus it is safe to parallelize the `check` methods. However, if an `XDocsReport` registers on line 19 in Figure 1, our hybrid system will dynamically enlarge the effect set of the `check` method. The previous effect set of `check` ({**ann** }) is unioned with the effect set of the method `reportBug` in `XDocsBugReporer`. The effect set of the method `reportBug` includes a write to a shared instance field. Thus the methods `check` in `MutableLock` and `FindBadCast` now conflict with each other. Therefore, our analysis determines that it is no longer safe to execute the `check` methods concurrently.

Thus, our hybrid analysis is able to expose concurrency when it is safe to do so whereas a purely static type-and-effect analysis would conservatively determine FindBugs as concurrency unsafe. To summarize, the main benefits of our hybrid type-and-effect system are:

- it is more precise compared to a fully static analysis resulting in greater concurrency, and
- its overhead is negligible and is amortized by the introduced concurrency.

We have proven several concurrency properties of our type-and-effect system. Most notably, we have proven the absence of races and determinism of programs using this type-and-effect system. Thus, users are guaranteed to avoid many complex issues that stem from concurrency.

To evaluate our approach, we have implemented this type-and-effect system in our language Pāñini [36], applied it to several applications, proven its key properties, and evaluated its performance. We have applied our type-and-effect system to scenarios where the static type-and-effect system was not able to give any concurrency benefits. These applications include FindBugs (110K LOC) [31], an e-mail filter (11K LOC) [2], a refactoring crawler (7K LOC) [17], a web crawler (16K LOC) [41] and a Genetic Algorithm (460 LOC) [48]. Our implementation shows almost linear speedup and negligible overhead. For FindBugs, we saw around 6x speedup in bug analysis code for 7 detectors, for a Genetic Algorithm, Pāñini gave 7x speedup, for the refactoring crawler, it measured about 3.5x speedup for 6 non-conflicting detectors in the detection, for the web crawler, it gave 15.7x speedup, and for the e-mail filter, we saw 1.7x speedup in spam detection code. This shows that our hybrid type-and-effect system provides considerable speedup. In summary, this paper makes the following contributions:

- a new hybrid type-and-effect system that facilitates concurrency in shared memory programs that use implicit-invocation design style;
- a precise dynamic semantics that uses the effect system to maximize concurrency;
- a soundness proof that our system guarantees no data races and no deadlocks; and
- a rigorous study on real world applications showing the applicability of our approach.

```
1 class FindBugs {
2   void analyze(Class c){
3     announce ClassAvailable(c);
4     // Further details elided
5   }
6 }

8 class XDocsReport {
9   Element root;
10  void init(){ register(this); }
11  when BugDetected do reportBug;
12  void reportBug(Bug bi){
13    // details elided
14    root.addElement(bi);
15  }
16 }
```

```
17 event ClassAvailable {
18   Class c;
19 }

21 class MutableLock {
22   void init(){ register(this); }
23   when ClassAvailable do check;
24   void check(Class c){
25     // details elided
26     if(/*...*/) {
27       announce BugDetected(
28                      new Bug(...));
29     }
30   }
31 }
```

```
32 event BugDetected {
33   Bug bi;
34 }

36 class FindBadCast {
37   void init(){ register(this); }
38   when ClassAvailable do check;
39   void check(Class c){
40     // details elided
41     if(/*...*/) {
42       announce BugDetected(
43                      new Bug(...));
44     }
45   }
46 }
```

Figure 3. Pāñini's implementation of FindBugs. Imperative code for implicit invocation is replaced by language features.

2. A Calculus with Event-based Concurrency

We present our type-and-effect system using a calculus with support for implicitly concurrent events. Our presentation builds on previous calculi [16, 46]. It formalizes language features that we have previously explored informally in our work on the Pāñini language [36]. Pāñini is an implicitly concurrent language, it does not feature any construct for spawning threads or for mutually exclusive access to shared memory. Rather, concurrent execution is facilitated by announcing events, using the **announce** expression, which may cause handlers to run concurrently. While previous work informally defined Pāñini [36], this calculus formalizes its definition as an expression language. Here, we describe the syntax in Figure 4 using the example from Section 1.

The program in Figure 3 is similar to the OO version in Figure 1 except that the code for implementing the observer pattern is replaced with Pāñini's constructs for declaring and announcing events. For example, the event type `BugDetected`, on lines 32-34, is used to decouple bug detectors from the concrete `Reporters`. Instead of registering with a certain bug detector, the concrete bug reporters register with an event. For example, on line 10, an `XDocsReport` instance could dynamically register with the event `BugDetected`. The code for calling the handler(s) in the bug detectors (e.g. `MutableLock` or `FindBadCast`) is replaced by an **announce** expression, on line 27 and 42 that notifies registered handlers.

Declarations. Pāñini features two new declarations compared to the Java language: event type (**event**) and binding declaration. An event has a name (p) and context variable declarations ($\overline{form}$). The over-bar denotes a finite ordered sequence and is used throughout this paper ($\overline{a}$ stands for $a_1 \ldots a_n$). For example, in Figure 3 on lines 17-19, an event of type `ClassAvailable` is defined. It has one context variable `c` of type `Class`, which denotes the class to be analyzed. These context variables are bound to actual values and made available to handlers when an event is fired. A binding declaration consists of two parts: an event type name and a method name. For example, on line 23, the class `MutableLock` declares a binding such that the `check` method is invoked whenever an event of type `ClassAvailable` is announced. This method may run concurrently with other handler methods.

$prog ::= \overline{decl}\ e$
$decl ::=$ **class** c **extends** d { $\overline{field}$ $\overline{meth}$ $\overline{binding}$ } | **event** p { $\overline{form}$ }
$field ::= c\ f$;
$meth ::= t\ m$ ($\overline{form}$) { e }
$t ::= c$ | **void**
$binding ::=$ **when** p **do** m ;
$form ::= c\ var$, where $var \neq$ **this**
$e ::=$ **new** c () | var | **null** | $e.m(\overline{e})$ | $e.f$ | $e.f = e$ | **cast** $c\ e$
  | $form = e$; e | e ; e | **register** (e) | **announce** p ($\overline{e}$)
where
$c, d \in \mathcal{C}$, the set of class names
$p \in \mathcal{P}$, the set of event type names
$f \in \mathcal{F}$, the set of field names
$m \in \mathcal{M}$, the set of method names
$var \in$ {**this**} $\cup\ \mathcal{V}$, $\mathcal{V}$ is the set of variable names

Added Syntax (used only in semantics) :

$e ::= loc$ | **yield** e | NullPointerException | ClassCastException
where $loc \in \mathcal{L}$, a set of locations

Figure 4. Pāñini's abstract syntax, based on [46].

Expressions. In Pāñini, handlers can register with events dynamically, e.g. line 10, 22 and 37. The syntax includes standard OO expressions for object allocation, variable binding and reference, **null** reference, method invocation, field access and update, type casting and sequence.

Concurrency in Pāñini. The **announce** expression in Pāñini is the source of concurrency. The expression **announce** `p` ($\overline{e}$) announces an event of type `p`, which may run any handlers that are applicable to p concurrently. In Figure 3 the body of the `analyze` method contains an **announce** expression on line 3. When the method signals this event, Pāñini looks for any applicable handlers. Suppose `MutableLock` and `FindBadCast` are registered with the event `ClassAvailable`. These handlers may execute concurrently, depending on whether they interfere with each other. Our hybrid type-and-effect system ensures that no conflicting handlers execute concurrently (details are in Section 3 and Section 4). After all the handlers are finished, the evaluation of the announce expression then continues on line 4. The announce expression, when signaled,

binds values to the event type's context variables. For example, when announcing event `ClassAvailable` on line 3, parameter `c` is bound to the context variable `c` on line 18. This binding makes the class to be analyzed available to handlers in the context variable `c`.

Intermediate Expressions. Four new expressions are added as shown in the bottom of Figure 4. The *loc* expression represents store locations. Following Abadi and Plotkin [4], we use the **`yield`** expression to model concurrency. The **`yield`** expression allows other tasks to run. Two exceptional final states, `NullPointerException` and `ClassCastException`, are reached when trying to access a field or a method from a **`null`** receiver or when an object is not a subtype of the casting type.

3. Type and Static Effect Computation

Our type-and-effect system has both a static and a dynamic part. The purpose of the static part is to compute the effects of handler methods, e.g. `check` in Figure 3. The purpose of the dynamic part is to use these statically computed effects to calculate the computational effects of **`announce`** expressions and to produce a concurrency-safe schedule. The type attributes used by both parts are defined in Figure 5.

$\theta ::= \text{OK}$	"program/decl/body types"
$\quad \mid \text{OK in c}$	"binding types"
$\quad \mid (t_1 \times \ldots \times t_n \to t, \rho)$ in c	"method types"
$\quad \mid (t, \rho)$	"expression types"
$\rho ::= \epsilon + \rho \mid \bullet$	"program effects"
$\epsilon ::=$ **read** $c\ f$	"read effect"
$\quad \mid$ **write** $c\ f$	"write effect"
$\quad \mid$ **ann** p	"announce effect"
$\quad \mid$ **reg**	"register effect"
$\pi, \Pi ::= \{I : t_I\}_{I \in K}$	"type environments"
where K is finite, $K \subseteq (\mathcal{L} \cup \{\mathbf{this}\} \cup \mathcal{V})$	

Figure 5. Type and effect attributes.

Compared to type systems that include events [46], new to our system are effects, e.g. the type attributes for expressions are represented as (t, ρ), the type of an expression (t) and its effect set (ρ). The effects are used to compute the potential conflicts between handlers. These effects include:

- read effect: a class and a field to be read;
- write effect: a class and a field to be written;
- announce effect: event an expression may announce and
- register effect: produced by a **`register`** expression.

For example, in Figure 3, before the program runs, the effect set of the method `check` in the class `MutableLock` is $\{$**ann** `BugDetected`$\}$ and the effect set of the method `reportBug` in the class `XDocsReport` is $\{$**`write`** `Element root`$\}$ (for simplicity, we assume that the method `addElement` only changes the field `root`). We have intentionally avoided tracking object instances to simplify this discussion. Instead, we focus on event registration and announcement, however, such extension is feasible.

The interference between effects is shown in Figure 6. Read effects do not conflict with each other. Write effects conflict with read and write effects accessing the same field of the same class. The announce effect is used later in the semantics. It serves as a place holder and does not conflict with other announce effects. Announce effects conflict with register effects, because the order of these two operations affects the set of handlers run during announcement (e.g. even if an event is fired, a handler will not run if it has not registered). Register effects interfere with read and write effects as well. After a handler registers with a certain event, the effect of some other handlers could be enlarged as well. Thus it could introduce cascading changes. Our hybrid system simply makes register effects conflict with any other effect.

Effects	**read**	**write**	**ann**	**reg**
read		×		×
write	×	×		×
ann				×
reg	×	×	×	×

Figure 6. Effect interference. × marks conflicting pairs

For example, before any handler registers with the event `BugDetected`, the effects of the methods `check` in both the class `MutableLock` and `FindBadCast` are $\{$**ann** `BugDetected`$\}$. Thus there is no conflict between them and it is safe to execute them concurrently. If an instance of the class `XDocsReport` registers, the effects of `check` becomes $\{$**ann** `BugDetected`, **`write`** `Element root`$\}$ in both these classes. Since these two write effects access the same field in the same class, the `check` methods now conflict with each other. The type system updates the announce effects of relevant handlers every time a handler registers with an event. Thus our system has more accurate information about the effects of the handlers than the pure static approaches when computing a schedule.

The type checking rules are shown in Figures 7 and 9. The notation $\nu' <: \nu$ means ν' is a subtype of ν. 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) as in Clifton's work [16]. The class table can be thought of as an implicit inherited attribute used by the rules and auxiliary functions. We require that top-level names in the program are distinct and that the inheritance relation on classes is acyclic. The typing rules for expressions use a simple type environment, Π, which is a finite partial mapping from locations *loc* or variable names *var* to a type and an effect set.

3.1 Top-level Declarations

The (T-PROGRAM) rule says that the entire program type checks if all the declarations type check and the expression e has any type t and any effect ρ. The (T-EVENT) rule says that an event declaration type checks if the types of all the context variables are declared properly.

The (T-CLASS) rule says that a class declaration type checks if all the following constraints are satisfied. First, all the newly declared fields are not fields of its super class (this is checked by the omitted auxiliary function *validF*). Next, its super class d is defined in the Class Table. Finally, all the declared methods and bindings type check.

The (T-METHOD) rule says that a method declaration type checks only if the return type is a class type (by the auxiliary function *isClass*(c), which searches CT to check whether the class c was declared. This function is used throughout this paper. If all the parameters have their corresponding declared types, the body of the method has type u and effect ρ (stored in CT); u is a subtype of the return type t. This rule uses an auxiliary function *override*, defined in Figure 8. It requires that the method has either a fresh name or the same type as the overridden superclass method [16]. This definition precludes overloading. In addition to standard conditions, this function enforces that the effect of an overriding method is the subset of the effect of the overridden method[1].

(T-PROGRAM)
$$\frac{(\forall decl_i \in \overline{decl} \;::\; \vdash decl_i : \text{OK}) \quad \vdash e : (t, \rho)}{\vdash \overline{decl}\; e : (t, \rho)}$$

(T-EVENT)
$$\frac{(\forall (t_i\; var_i) \in \overline{t\; var;} \;::\; \textit{isClass}(t_i))}{\vdash \textbf{event}\; p\; \{\overline{t\; var;}\} : \text{OK}}$$

(T-CLASS)
$$\frac{\textit{validF}(\overline{t\; f}, d) \quad (\forall b \in \overline{binding} \;::\; \vdash b : \text{OK in } c) \quad \textit{isClass}(d) \quad (\forall meth_j \in \overline{meth} \;::\; \vdash meth_j : (t_j, \rho_j)\; in\; c)}{\vdash \textbf{class}\; c\; \textbf{extends}\; d\; \{\overline{t\; f};\; \overline{meth}\; \overline{binding}\} : \text{OK}}$$

(T-BINDING)
$$\frac{\begin{array}{c} CT(p) = \textbf{event}\; p\; \{t_1\; var_1, \ldots, t_n\; var_n\} \\ (c_1, t, m\,(\overline{t'\; var'})\;\{e\}, \rho) = \textit{findMeth}(c, m) \\ \pi = \{var_1 : t_1, \ldots, var_n : t_n\} \\ (\forall\, (t'_i\; var'_i) \in \overline{t'\; var'} \;::\; \pi(var'_i) <: t'_i) \end{array}}{\vdash \textbf{when}\; p\; \textbf{do}\; m \;: \text{OK in } c}$$

(T-METHOD)
$$\frac{\begin{array}{c} override(m, c, (t_1 \times \ldots \times t_n \rightarrow t, \rho)) \\ (\forall i \in \{1..n\} :: \textit{isClass}(t_i)) \quad u <: t \\ \textit{isClass}(t) \quad (var_1 : t_1, \ldots, var_n : t_n, \textbf{this} : c) \vdash e : (u, \rho) \end{array}}{\vdash t\; m(t_1\; var_1, \ldots, t_n\; var_n)\{e\} : (t_1 \times \ldots \times t_n \rightarrow t, \rho) \text{ in } c}$$

Figure 7. Type-and-effect rules for declarations [16, 46].

$$\frac{\begin{array}{c} CT(c) = \textbf{class}\; c\; \textbf{extends}\; d\; \{\ldots\; meth_1 \ldots meth_p\} \\ \nexists i \in \{1..p\} \cdot meth_i = t\; m(t_1\; var_1, \ldots, t_n\; var_n)\{e\} \\ override(m, d, (t_1 \times \ldots \times t_n \rightarrow t, \rho)) \end{array}}{override(m, c, (t_1 \times \ldots \times t_n \rightarrow t, \rho))}$$

$$\frac{(c_1, t, m\,(\overline{t'\; var'})\;\{e\}, \rho') = \textit{findMeth}(d, m) \qquad \rho \subseteq \rho'}{override(m, d, (t_1 \times \ldots \times t_n \rightarrow t, \rho))}$$

$$override(m, Object, (t_1 \times \ldots \times t_n \rightarrow t, \rho))$$

Figure 8. Auxiliary function for checking overriding.

[1] In practice, we enlarge the effect set of the method in the super class such that the effect of the overriding method is a subset of its super class. An alternative could be to raise a type error.

The (T-BINDING) rule says that a binding declaration type checks if the named method is properly defined; all the context variables are subtypes of their corresponding declared types in the method; the named event type is declared properly. It uses the auxiliary function *findMeth*. This function looks up the method m, starting from the type of the expression, looking in super classes, if necessary.

3.2 Expressions

The type rules for the expressions are shown in Figure 9. The rules for object-oriented expressions are mostly standard, except for the addition of effects in type attributes. The (T-NEW) rule ensures that the class c being instantiated was declared. This expression has type c and empty effect. The (T-GET) rule says that a field access expression returns the type of the field of the class, the effects of it will be the effect of the object expression plus a read effect. The auxiliary function *typeOfF* used in this rule finds the type of a field. The (T-SET) rule says that a field assignment expression type checks if the object expression is of a class type and the type of the assignment expression e_2 is a subtype of the type of the field of the class. The effects will be the union of the effects of its two subexpressions plus one **write** effect. The rule for null expression, (T-NULL) is also standard.

(T-NEW)
$$\frac{\textit{isClass}(c)}{\Pi \vdash \textbf{new}\; c() : (c, \{\})}$$

(T-CAST)
$$\frac{\textit{isClass}(c) \quad \Pi \vdash e : (t', \rho)}{\Pi \vdash \textbf{cast}\; c\; e : (c, \rho)}$$

(T-SEQUENCE)
$$\frac{\Pi \vdash e_1 : (t_1, \rho) \quad \Pi \vdash e_2 : (t_2, \rho')}{\Pi \vdash e_1; e_2 : (t_2, \rho \cup \rho')}$$

(T-YIELD)
$$\frac{\Pi \vdash e : (t, \rho)}{\Pi \vdash \textbf{yield}\; e : (t, \rho)}$$

(T-VAR)
$$\frac{\Pi(var) = t}{\Pi \vdash var : (t, \{\})}$$

(T-DEFINE)
$$\frac{\textit{isClass}(c) \quad \Pi \vdash e_1 : (t_1, \rho) \quad \Pi, var : c \vdash e_2 : (t_2, \rho') \quad t_1 <: c}{\Pi \vdash c\; var = e_1; e_2 : (t_2, \rho \cup \rho')}$$

(T-SET)
$$\frac{\Pi \vdash e : (c, \rho) \quad \textit{typeOfF}(c, f) = t \quad \Pi \vdash e' : (t', \rho') \quad t' <: t}{\Pi \vdash e.f = e' : (t', \rho \cup \rho' \cup \{\textbf{write}\; c\; f\})}$$

(T-NULL)
$$\frac{\textit{isType}(t)}{\Pi \vdash \textbf{null} : (t, \{\})}$$

(T-CALL)
$$\frac{\begin{array}{c} (c_1, t, m\,(t_1\; var_1, \ldots, t_n\; var_n)\;\{e_{n+1}\}, \rho) = \textit{findMeth}(c_0, m) \\ \Pi \vdash e_0 : (c_0, \rho_0) \quad (\forall\, i \in \{1..n\} \;::\; \Pi \vdash e_i : (t'_i, \rho_i) \;\wedge\; t'_i <: t_i) \end{array}}{\Pi \vdash e_0.m(e_1, \ldots, e_n) : (t, \rho \cup \bigcup_{i=1}^{n} \rho_i \cup \rho_0)}$$

(T-GET)
$$\frac{\Pi \vdash e : (c, \rho) \quad \textit{typeOfF}(c, f) = t}{\Pi \vdash e.f : (t, \rho \cup \{\textbf{read}\; c\; f\})}$$

(T-REGISTER)
$$\frac{\Pi \vdash e : (t, \rho) \quad \textit{isClass}(t)}{\Pi \vdash \textbf{register}\,(e) : (t, \rho \cup \{\textbf{reg}\})}$$

(T-ANNOUNCE)
$$\frac{\begin{array}{c} CT(p) = \textbf{event}\; p\; \{t_1\; var_1;\; \ldots\; t_n\; var_n;\} \\ (\forall\, i \in \{1..n\} \;::\; \Pi \vdash e_i : (t'_i, \rho_i) \;\wedge\; t'_i <: t_i) \end{array}}{\Pi \vdash \textbf{announce}\; p\; (e_1, \ldots, e_n) : (\textbf{void}, \{\textbf{ann}\; p\} \cup \bigcup_{i=1}^{n} \rho_i)}$$

Figure 9. Type and effect rules for expressions[16, 46].

The (T-CAST) rule says that for a cast expression, the cast type must be a class type, and its effect is the same as the expression's. The (T-SEQUENCE) rule states that the sequence expression has same type as the last expression and its effects are the union of the two expressions. The (T-YIELD) rule says that a **yield** expression has the same type and same effect as the expression e. The (T-VAR) rule checks that var is in the environment. The (T-DEFINE) rule for declaration expressions is similar to the sequence expression except that the initial expression should be a subtype of the type of the new variable. Also, the type of the variable is placed in the environment. Finally, the sequence expression type checks if both left and right expressions type check and has the combined effect of both expressions.

The (T-REGISTER) rule says that a register expression has the same type as its object expression and the effects will be the effects of its object expression plus one register effect. For register, we do not include information about the event (e.g. **reg** p), because it will not be more accurate: after a handler registers with an event p, effects of handlers for other events could be enlarged as well. Thus, we assume that a register effect conflicts with all other effects. The (T-ANNOUNCE) rule ensures that the event was declared and the actual parameters are subtypes of the context variables in the event declaration. The entire expression has the type **void**. The effects of the announce expression will be the union of all the parameter expressions' effects plus one announcement effect.

The (T-CALL) is similar to the announce expression. This rule says that for a method call expression it finds the method in the CT using the auxiliary function `findMeth` (same as that used for checking binding declarations previously) and this method is declared either in its own class or its super class. Each actual argument expression is of subtype of corresponding parameter type. This method call expression has the same type as the return type of the method.

$$isClass(t) = t \in dom(CT) \land CT(t) = \textbf{class}\ t \ldots$$

$$isType(t) = isClass(t) \lor t = \textbf{void}$$

$$fieldsOf(c) = \{t_i\} \cup fieldsOf(c') \quad \textbf{where}\ CT(c) = \textbf{class}\ c\ \textbf{extends}\ c'\{t_1\ f_1; \ldots t_n\ f_n; \ldots\}$$
$$validF(\overline{t\ f}, c) = \forall i \in \{1..n\}\ ::\ isClass(t_i) \land f_i \notin dom(fieldsOf(c))$$

Figure 10. Auxiliary functions used in type rules.

4. Pāñini's Dynamic Semantics with Effect-based Task Scheduling

Here we give a small-step operational semantics for Pāñini. The main novelty in our semantics is the integration of an effect system with a scheduling algorithm that produces safe execution, while maximizing concurrency for programs that use implicit-invocation mechanisms.

Evaluation relation: $\hookrightarrow : \Sigma \dashrightarrow \Sigma$
Domains:

Σ	$::= \langle \psi, \mu, \gamma \rangle$	"Program Configurations"
ψ	$::= \langle e, \tau \rangle + \psi \mid \bullet$	"Task Queue"
τ	$::= \langle n, \{n_k\}_{k \in K} \rangle$ where $n,\ n_k \in \mathbb{N}$ and K is finite	"Task Dependencies"
μ	$::= \{loc \mapsto o_k\}_{k \in K}$, where K is finite	"Stores"
v	$::= \textbf{null} \mid loc$	"Values"
o	$::= [c.F]$	"Object Records"
F	$::= \{f_k \mapsto v_k\}_{k \in K}$, where K is finite	"Field Maps"
γ	$::= \{p \mapsto \delta\}$	"Event map"
δ	$::= \zeta + \delta \mid \bullet$	"Handler Hierarchy"
ζ	$::= \iota + \zeta \mid \bullet$	"Handler List"
ι	$::= \langle loc, m, \rho \rangle$	"Handler Configuration"

Evaluation contexts:

$$\begin{array}{ll} \mathbb{E} & ::= - \mid \mathbb{E}.m(e\ldots) \mid v.m(v\ldots\mathbb{E}\,e\ldots) \mid \textbf{cast}\ t\ \mathbb{E} \\ & \mid \mathbb{E}.f \mid \mathbb{E}.f\texttt{=}e \mid v.f\texttt{=}\mathbb{E} \mid t\ var\texttt{=}\mathbb{E};\ e \mid \mathbb{E};\ e \\ & \mid \textbf{announce}(v\ldots\mathbb{E}\,e\ldots) \mid \textbf{register}(\mathbb{E}) \end{array}$$

Figure 11. Domains, and evaluation contexts used in the semantics, based on [46].

Domains. The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 11. The rules and auxiliary functions all make use of an implicit attribute CT, the program's declarations.

A configuration consists of a task queue ψ, a global store μ, and a global event map γ. The store μ is a mapping from locations (loc) to objects (o). The map γ maps event p to a event hierarchy δ. Each event hierarchy consists of a list ζ of list. Each item ι is a tuple consists of the handler object loc, the handler method m and the effect set ρ for this handler. The task queue ψ consists of an ordered list of task configurations $\langle e, \tau \rangle$. This configuration consists of an expression e running in that task and the corresponding task dependencies τ. This expression e serves as the remaining evaluation to be done for the task. Pāñini orders the handlers as a hierarchy. Handlers in each level of the event hierarchy depends on previous levels. These handlers do not get executed until handlers in the previous levels are done executing.

The task dependencies are used to record the identity of the current task (n) and a set of identities for other tasks on which this task depends. We call this set the dependence set of the task. A task t depends on another task t' if 1) t's effect set conflicts with the effect set of t' and t' registered before t or 2) if t' is a handler task for an **announce** expression t is evaluating. In the semantics, a task is never scheduled unless all the tasks it depends on are finished.

An object record o consists of a class name c and a field record F. A field record is a mapping from field names f to values v. A value v may either be **null** or a location loc, which have standard meanings.

Evaluation Contexts. We present the semantics as a set of evaluation contexts $\mathbb{E}$ (Figure 11) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context [59]. This avoids the need for writing out standard recursive rules and clearly presents the order of evaluation. The language uses a call-

by-value evaluation strategy. The initial configuration of a program with a main expression e is $\langle\langle e, \langle 0, \emptyset\rangle\rangle, \bullet, \bullet\rangle$.

Semantics for Object-oriented Expressions. The rules for OO expressions are given in Figure 12. These are mostly standard and adopted from Ptolemy's semantics [46]. The operator $\oplus$ is an overriding operator for finite functions, i.e. if $\mu' = \mu \oplus \{loc \mapsto v\}$, then $\mu'(loc') = v$ if $loc' = loc$, otherwise $\mu'(loc') = \mu(loc')$.

(SEQUENCE)
$$\langle\langle \mathbb{E}[v; e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, e], \tau\rangle + \psi, \mu, \gamma\rangle$$

(NEW)
$$\frac{loc \notin dom(\mu) \qquad \mu' = \{loc \mapsto [c.\{f \mapsto \texttt{null} \mid (t\ f) \in \mathit{fieldsOf}(c)\}]\} \oplus \mu}{\langle\langle \mathbb{E}[\texttt{new}\, c()], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, loc], \tau\rangle + \psi, \mu', \gamma\rangle}$$

(CALL)
$$\frac{(c', t, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}, \rho) = \mathit{findMeth}(c, m) \qquad [c.F] = \mu(loc) \qquad e' = [loc/\texttt{this}, v_1/var_1, \ldots, v_n/var_n]e}{\langle\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, e'], \tau\rangle + \psi, \mu, \gamma\rangle}$$

(DEFINE)
$$\frac{e' = [v/var]e}{\langle\langle \mathbb{E}[t\ var = v; e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, e'], \tau\rangle + \psi, \mu, \gamma\rangle}$$

(CAST)
$$\frac{[c'.F] = \mu(loc) \qquad c' <: c}{\langle\langle \mathbb{E}[\texttt{cast}\, c\ loc], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, loc], \tau\rangle + \psi, \mu, \gamma\rangle}$$

(GET)
$$\frac{\mu(loc) = [c.F] \qquad v = F(f)}{\langle\langle \mathbb{E}[loc.f], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, v], \tau\rangle + \psi, \mu, \gamma\rangle}$$

(SET)
$$\frac{[c.F] = \mu(loc) \qquad \mu' = \mu \oplus (loc \mapsto [c.F \oplus (f \mapsto v)])}{\langle\langle \mathbb{E}[loc.f = v], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\, v], \tau\rangle + \psi, \mu', \gamma\rangle}$$

Figure 12. Semantics of OO expressions in Pāñini

One difference stems from the concurrency and store models in Pāñini. The use of the intermediate expression **`yield`** in all these rules serves to allow other tasks to run.

The (NEW) rule creates a new object and initializes its fields to null. It then creates a record with a mapping from a reference to this newly created object. The *fieldsOf* function, in Figure 13, returns a map from all the fields defined in the class and its supertypes to the types of those fields.

$$\frac{CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{t_1\ f_1 \ldots t_n\ f_n\ \overline{meth}\ \overline{binding}\} \qquad \mathit{fieldsOf}(d) = Ft'}{\mathit{fieldsOf}(c) = \{f_i \mapsto t_i\ ::\ i \in \{1 \ldots n\}\} \cup Ft'}$$

$$\mathit{fieldsOf}(Object) = \{\}$$

Figure 13. Auxiliary function *fieldsOf*.

The (CALL) rule acquires the method signature using the auxiliary function *findMeth* (defined in Figure 14). It uses dynamic dispatch, which starts from the dynamic class (c) of the record, and may look up the super class of the object if needed. The method body is to be evaluated with the arguments substituted by the actual values as well as the `this` variable by *loc*. The entire substituted method body is then put inside a yield expression to model concurrency, which will be discussed later.

$$\frac{CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ meth_1 \ldots meth_p\ \overline{binding}\} \qquad \exists i \in \{1 \ldots p\}\ ::\ meth_i = (t, \rho, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\})}{\mathit{findMeth}(c, m) = (c, t, m(t_1\ var_1, \ldots, t_n\ var_n), \rho)}$$

$$\frac{CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ meth_1 \ldots meth_p\ \overline{binding}\} \qquad \nexists i \in \{1 \ldots p\}\ ::\ meth_i = (t, \rho, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}) \qquad \mathit{findMeth}(d, m) = l}{\mathit{findMeth}(c, m) = l}$$

Figure 14. Auxiliary function *findMeth*.

The (SEQUENCE) rule says that the current task may yield control after the evaluation of the first expression. The (CAST) rule is used only when the *loc* is a valid record in the store and when the type of object record pointed to by *loc* is subtype of the cast type. The (DEFINE) rule allows for local definitions. It binds the variable to the value and evaluates the subsequent expressions with the new binding.

The (GET) rule gets an object record from the store and retrieves the corresponding field value as the result. The semantics for (SET) first fetches the object from the store and overrides the field using the overriding operator $\oplus$.

Semantics for Yielding Control. In Pāñini's semantics, the running task may implicitly relinquish control to other tasks. The rules for yielding control are given in Figure 15.

(YIELD)
$$\frac{\langle e', \tau'\rangle + \psi' = active(\psi + \langle \mathbb{E}[e], \tau\rangle)}{\langle\langle \mathbb{E}[\texttt{yield}\, e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle e', \tau'\rangle + \psi', \mu, \gamma\rangle}$$

(TASK-END)
$$\frac{\langle e', \tau'\rangle + \psi' = active(\psi) \qquad \psi \neq \bullet}{\langle\langle v, \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle e', \tau'\rangle + \psi', \mu, \gamma\rangle}$$

(YIELD-DONE)
$$\langle\langle \mathbb{E}[\texttt{yield}\, e], \tau\rangle + \bullet, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[e], \tau\rangle + \bullet, \mu, \gamma\rangle$$

Figure 15. Semantics of yielding control in Pāñini

The (YIELD) rule puts the current task configuration to the end of the task-queue and starts evaluating the next active task configuration from this queue. Finding an active task is done by the auxiliary function $active$ (shown in Figure 16). It returns the top most task configuration in the queue that could be run. A task configuration is ready to run if all the tasks in its dependence set are done (evaluated to a single value v). The (YIELD-DONE) rule is applied when there is no

$$\begin{array}{ll}
active(\langle e, \tau\rangle + \psi) = \langle e, \tau\rangle + \psi & \textbf{if } intersect(\tau, \psi) = false \\
active(\langle e, \tau\rangle + \psi) = active(\psi + \langle e, \tau\rangle) & \textbf{if } intersect(\tau, \psi) = true \\
\\
intersect(\emptyset, \psi) = false & \\
intersect(\{n\} \cup \tau, \psi) = true & \textbf{if } inQueue(n, \psi) = true \\
intersect(\{n\} \cup \tau, \psi) = intersect(\tau, \psi) & \textbf{if } inQueue(n, \psi) = false \\
\\
inQueue(n, \bullet) = false & \\
inQueue(n, \langle e, \langle n, \{n_k\}\rangle\rangle + \psi) = true & \\
inQueue(n, \langle e, \langle n', \{n_k\}\rangle\rangle + \psi) = inQueue(n, \psi) & \textbf{if } n \neq n'
\end{array}$$

Figure 16. Functions for finding a nonblocked task.

other task configuration in the queue. It continues to evaluate

the current configuration. The (TASK-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.

Semantics for Event registration. The semantics for subscribing to an event is given in Figure 17. The (REGISTER) rule makes use of the function $updateHierarchy$ to insert the handler into the event hierarchy (γ). This auxiliary function, defined in the appendix, also updates the effect sets for all tasks which announce the event this handler is interested in. This update of the effect sets happens repeatedly until a fixed point is reached.

(REGISTER)

$$\frac{loc \notin \gamma \qquad updateHierarchy(loc, \gamma, \mu) = \gamma'}{\langle\langle \mathbb{E}[\texttt{register}(loc)], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\ loc], \tau\rangle + \psi, \mu, \gamma'\rangle}$$

(ANNOUNCE)

$$\frac{\begin{array}{c}\texttt{event}\ p\{t_1\ var_1, \ldots, t_n\ var_n\} = CT(p) \\ \tau = \langle id, I'\rangle \qquad \psi' = \psi + \psi'' \qquad \tau' = \langle id, I\rangle \\ \nu = (v_1, \ldots, v_n) \qquad \langle \psi'', I\rangle = spawn(p, \psi, \gamma, \nu, \mu)\end{array}}{\begin{array}{c}\langle\langle \mathbb{E}[\texttt{announce}\ p\ (v_1, \ldots, v_n)], \tau\rangle + \psi, \mu, \gamma\rangle \\ \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield null}], \tau'\rangle + \psi', \mu, \gamma\rangle\end{array}}$$

Figure 17. Semantics of registration and announcement

Semantics for announcing an event. The semantics for signaling events is shown in Figure 17. The (ANNOUNCE) rule takes the relevant event declaration from CT (the program's list of declarations) and creates a list of actual parameters (ν). This list of actual parameters (ν) is used by the auxiliary function $spawn$ shown in Figure 18 (with other helper functions in Figure 19). The (ANNOUNCE) rule resorts to the auxiliary function $spawn$ for two tasks: 1) finding the handlers registered for the corresponding event; 2) for executing them according to the hierarchy (computed at registration) to guarantee safety and maximize concurrency. It can then safely put the handler configurations (returned from the auxiliary function $spawn$) into the queue. Note that the task dependencies, τ may be safely dropped. This is because if there were conflicts, t could not have been executed until the conflicting handlers completed. Thus, the dependencies are no longer relevant. The auxiliary function $concat$ is used in several other auxiliary functions. It combines the contents in the two lists, which are the inputs to this function. The first task is done by the function $spawn$. It searches the program's global list of handlers (γ) for applicable handlers, using auxiliary functions $hfind$(finds method body for each handler), $hmatch$(finds the class for a specific handler), and $match$(finds the correct binding from a list of bindings) [46].

The second task is done by the functions $buildconfs$ (Figure 19) and $buildconf$. They create task configurations for handlers. $buildconf$ binds the context variables (of the event type) with the values (ν), computes a unique id for each handler task, and configures the dependence set of this handler.

$$\begin{array}{l}
spawn(p, \psi, \gamma, \nu, \mu) = buildconfs(H, \psi, \nu, \bullet, \gamma, \mu) \\
\quad \textbf{where}\ H = hfind(\gamma, p, \mu) \\[1ex]
hfind(\bullet, p, \mu) = \bullet \\
hfind(loc + \gamma, p, \mu) = hfind(\gamma, p, \mu) \\
\qquad \textbf{if}\ \mu(loc) = [c.F] \ \wedge\ hmatch(c, p, CT) = \bullet \\
hfind(loc + \gamma, p, \mu) = concat(hfind(\gamma, p, \mu), \langle loc, m\rangle) \\
\qquad \textbf{if}\ \mu(loc) = [c.F] \ \wedge\ hmatch(c, p, CT) = m \\[1ex]
hmatch(c, p, \bullet) = \bullet \\
hmatch(c, p, (\texttt{event}\ p\{\ldots\}) + CT') = hmatch(c, p, CT') \\
hmatch(c, p, (\texttt{class}\ c' \ldots) + CT') = hmatch(c, p, CT') \quad \textbf{if}\ c \neq c' \\
hmatch(c, p, ((\texttt{class}\ c\ \texttt{extends}\ d\ \ldots\ binding_1 \ldots binding_n) + CT')) \\
\quad = excl(match((binding_n + \ldots + binding_1), p), hmatch(d, p, CT)) \\
\qquad \textbf{where}\ excl(\bullet, H) = H\ \textbf{and}\ excl(e, H) = e \\[1ex]
match(\bullet, p) = \bullet \\
match((\texttt{when}\ p'\ \texttt{do}\ m) + B, p) = match(H, p) \quad \textbf{if}\ p' \neq p \\
match((\texttt{when}\ p\ \texttt{do}\ m) + B, p) = m
\end{array}$$

Figure 18. Functions for creating task configurations.

$$\begin{array}{l}
buildconfs(\bullet, \psi, \nu, H', \gamma, \mu) = (\bullet, \bullet) \\
buildconfs(\langle loc, m\rangle + H, \psi, \nu, H', \gamma, \mu) \\
\qquad = (\langle e, \langle id, I\rangle\rangle + \psi', concat(id, I')) \\
\quad \textbf{where}\ \langle e, \langle id, I\rangle\rangle = buildconf(loc, m, \psi, \nu, H', \gamma, \mu)\ , \\
\quad H'' = H' + \langle loc, m\rangle, \quad (\psi', I') = buildconfs(H, \psi, \nu, H'', \gamma, \mu) \\[1ex]
buildconf(loc, m, \psi, \nu, H, \gamma, \mu) \ = \\
\qquad \textbf{let}\ e' = [\texttt{this}/loc, var_1/v_1, \ldots, var_n/v_n]e \ \ \textbf{in}\ \langle e', \langle id, I\rangle\rangle \\
\quad \textbf{where}\ loc = [c.F]\ , \quad \nu = (v_1, \ldots, v_n)\ , \\
\quad (c', t, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}, \ldots) = findMeth(c, m), \\
\quad I = preds(loc, m, H, id' + 1, \gamma, \mu)\ , \quad id = fresh() \\[1ex]
preds(loc, m, \bullet, n, \gamma, \mu) = \bullet \\
preds(loc, m, \langle loc_1, m_1\rangle + H, n, \gamma, \mu) = preds(loc, m, H, n+1, \gamma, \mu) \\
\quad \textbf{if}\ true = indep(update(\rho, \gamma, \mu), update(\rho', \gamma, \mu)) \\
\quad \textbf{where}\ loc = [c.F], \qquad (c', t, m \ldots, \rho) = findMeth(c, m), \\
\quad loc_1 = [c_1.F], \qquad (c_1', t_1, m_1 \ldots, \rho') = findMeth(c_1, m_1) \\
preds(loc, m, \langle loc_1, m_1\rangle + H, n, \gamma, \mu) = concat(n, preds(loc, m, H, n+1, \gamma, \mu)) \\
\quad \textbf{if}\ false = indep(update(\rho, \gamma, \mu), update(\rho', \gamma, \mu)) \\
\quad \textbf{where}\ loc = [c.F], \qquad (c', t, m \ldots, \rho) = findMeth(c, m), \\
\quad loc_1 = [c_1.F], \qquad (c_1', t_1, m_1 \ldots, \rho') = findMeth(c_1, m_1) \\[1ex]
update(\bullet, \gamma, \mu) = \bullet \\
update(\langle \texttt{read}\ c\ f\rangle + \rho, \gamma, \mu) = concat(\langle \texttt{read}\ c\ f\rangle, update(\rho, \gamma, \mu)) \\
update(\langle \texttt{write}\ c\ f\rangle + \rho, \gamma, \mu) = concat(\langle \texttt{write}\ c\ f\rangle, update(\rho, \gamma, \mu)) \\
update(\langle \texttt{create}\rangle + \rho, \gamma, \mu) = concat(\langle \texttt{create}\rangle, update(\rho, \gamma, \mu)) \\
update(\langle \texttt{reg}\rangle + \rho, \gamma, \mu) = concat(\langle \texttt{reg}\rangle, update(\rho, \gamma, \mu)) \\
update(\langle \texttt{ann}\rangle + \rho, \gamma, \mu) \\
\quad = concat(\ getE(hfind(\gamma, p, \mu), \gamma, \mu), update(\rho, \gamma, \mu))
\end{array}$$

Figure 19. Functions for building handler configurations.

These task configurations are used to run the handler bodies and are appended to the end of the queue ψ. The auxiliary function $fresh$ is used for giving the newly-born task a fresh ID.

The auxiliary function $preds$ is used to find the dependence set for a task t. It first calls another function $update$ to update the effects of the task. It uses the $findMeth$ to retrieve the effects of methods from CT. The function $update$ is used to model the effect enlargement discussed in the beginning of the section, i.e. it is the dynamic phase of the hybrid system. This function searches the handler queue γ for applicable handlers, registered for events that this current task could

signal, and unions their effect sets with the effect set of this task t (e.g. if t may announce an event p, then the effect of all the handlers registered for event p will be used). Pāñini does this to get more accurate information about the potential effect sets of a task to reduce false conflicts. The function $indep$ (shown in Figure 20) is used to actually compare the effects to check whether they conflict with each other. The table in Figure 6 is used to compare effects.

$$size(\bullet) = 0 \qquad size(\langle loc, m\rangle + H) = 1 + size(H)$$

$$getE(\bullet, \gamma, \mu) = \bullet$$
$$getE(\langle loc, m\rangle + H, \gamma, \mu) = concat(update(\rho, \gamma, \mu), getE(H, \gamma, \mu))$$
$$\textbf{where } loc = [c.F], \quad (c', t, m \ldots, \rho) = findMeth(c, m)$$

$$indep(\bullet, \rho) = true$$
$$indep(\epsilon + \rho', \rho) = indep(\rho', \rho) \qquad \textbf{if } true = differ(\epsilon, \rho)$$
$$indep(\epsilon + \rho', \rho) = false \qquad \textbf{if } false = differ(\epsilon, \rho)$$

$$differ(\epsilon, \bullet) = true$$
$$differ(\epsilon, \epsilon' + \rho) = differ(\epsilon, \rho) \qquad \textbf{if } \epsilon \text{ and } \epsilon' \text{ have no conflicts}$$
$$differ(\epsilon, \epsilon' + \rho) = false \qquad \textbf{if } \epsilon \text{ and } \epsilon' \text{ have conflicts}$$

$$concat(\bullet, L') = L' \qquad concat(l + L, L') = l + concat(L, L')$$

Figure 20. Miscellaneous helper functions.

5. Evaluation: Safety Properties

We now show key properties of our type-and-effect system.

5.1 Deadlock Freedom

The first property of our type-and-effect system is that it provides deadlock freedom. We now state and provide a proof sketch of this property.

Definition [Predecessor] A task t_1 is a predecessor of another task t_2 (denoted as $t_1 \leq t_2$) , if either 1) t_2 announces an event and t_1 is a handler for the event (a task, which announces an event, has to wait for all the handlers to finish, as described in Section 4), or 2) handlers h_1 and h_2, corresponding to t_1 and t_2 respectively, are handlers for the same event, h_1 registers earlier than h_2, and the effect set of h_2 conflicts with the effect set of h_1.

Definition [Blocked Configurations.] A task configuration $\langle e, \tau\rangle$ in a program configuration $\langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle$ may block if any one of its predecessor tasks is still in execution.

Theorem 5.1. [Liveness.] *Let $\sigma = \langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle$ be a program configuration, where e is a well-typed expression, τ is task dependencies, μ is the store, ψ is a task queue and γ is a handler queue. Then either $\langle e, \tau\rangle$ is not blocked or there is some task configuration $\langle e', \tau'\rangle \in \psi$ that is not blocked.*

Proof Sketch: Construct a tree of the current tasks where parent nodes, t, have announced events, e, and the handlers of e, T_e, form the children of t. This relationship is denoted as $succ(t, t')$ where $t' \in T_E$. Nodes in a lower level will never depend on nodes in the above levels. That is, $\forall t_1, t_2$ s.t. $succ(t_1, t_2) \Rightarrow \neg t_1 \leq t_2$. Children will never depend on their parents by definition of the $spawn$ function (the *buildHier* doesn't take as a parameter higher levels of the hierarchy, thus, no dependencies may be created). A node, t_3, may depend on its sibling, t_2, $(succ(t_1, t_2) \wedge succ(t_1, t_3))$ denoted as $t_2 \leq t_3$, if t_3's effect set conflicts with t_2's effect set and t_2 registers first. Since one of these handlers must register first, there can not be a circular dependency. That is, $t_2 \leq t_3 \Rightarrow \neg t_3 \leq t_2$ where t_2 and t_3 are siblings. Finally, leaf nodes, $\forall p$ s.t. $\nexists p'$ s.t. $succ(p, p')$, have no children and thus may depend on nothing except their siblings. If there is no conflict between the children then they may run concurrently. If there exists a conflict between the children, at least the first of the remaining handlers to register may be run. This is because there may be no circular dependencies among children as shown above and this handler registered earlier than all remaining handlers. Thus, in the lowest level of the tree (leaves), there is at least one task (the handler in this level that registered earlier than any of its siblings) that does not block. Therefore, a well typed Pāñini program does not deadlock.∎

5.2 Deterministic Semantics (Data Race Freedom)

Another property of our type-and-effect system is that it provides deterministic semantics. This is accomplished by ensuring data race freedom (as defined by Bocchino [10, pp. 16]). The statement and proof of this property builds on Welc's work [57].

Definition [Schedule.] A schedule (χ) is a sequence of read, write, announce and register operations performed during the evaluation of a program. More precisely, $\chi ::= \overline{\eta}$, where $\eta ::= (\texttt{rd}, n, loc, f) \mid (\texttt{wt}, n, loc, f) \mid (\texttt{an}, n, p) \mid (\texttt{rg}, n)$ and n is a task ID.

$$\frac{\langle\langle \mathbb{E}[loc.f], \langle t, \ldots\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \Sigma \qquad \chi' = \chi + (\texttt{rd}, t, loc, f)}{\chi \hookrightarrow \chi'}$$

$$\frac{\langle\langle \mathbb{E}[loc.f = v], \langle t, \ldots\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \Sigma \qquad \chi' = \chi + (\texttt{wt}, t, loc, f)}{\chi \hookrightarrow \chi'}$$

$$\frac{\begin{array}{c}\langle\langle \mathbb{E}[\texttt{announce}\ p\ (v_1, \ldots)], \langle t, \ldots\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \Sigma \\ \chi' = \chi + (\texttt{an}, t, p)\end{array}}{\chi \hookrightarrow \chi'}$$

$$\frac{\langle\langle \mathbb{E}[\texttt{register}(loc)], \langle t, \ldots\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \Sigma \qquad \chi' = \chi + (\texttt{rg}, t)}{\chi \hookrightarrow \chi'}$$

Definition [Schedule Safety.] A schedule χ is safe if and only if:

1. an access to a field of an object $o.f$ performed by a predecessor should not witness a write to $o.f$ by its successor[2]($ssafe$);
2. a write to $o.f$ by a predecessor should be visible to the first access to that field by its successor ($psafe$);

[2] A task t is a successor of t' (denoted as $t' \leq t$.) if t' is a predecessor of t.

3. an event announcement by a predecessor should not notify handlers registered by its successor (*rsafe*) and
4. an event announcement by a successor should notify handlers registered by its predecessor (*asafe*).

$$\frac{(\mathtt{wt}\,,t',loc,f),(\mathtt{rd}\,,t',loc,f)\notin\chi' \quad t'\leq t}{ssafe(\chi+(\mathtt{wt}\,,t,loc,f)+\chi')}$$

$$\frac{(\mathtt{wt}\,,t',loc,f),(\mathtt{rd}\,,t',loc,f)\notin\chi \quad t\leq t'}{psafe(\chi+(\mathtt{wt}\,,t,loc,f)+\chi')}$$

$$\frac{(\mathtt{an}\,,t',p),(\mathtt{rg}\,,t')\notin S' \quad t'\leq t}{rsafe(\chi+(\mathtt{rg}\,,t)+\chi')} \qquad \frac{(\mathtt{rg}\,,t')\notin\chi \quad t\leq t'}{asafe(\chi+(\mathtt{an}\,,t,p)+\chi')}$$

The first two conditions are roughly the same as in Welc's work [57], while the last two are necessary to ensure that handlers only handle appropriate events.

Definition [Permute.] Schedule S is a permutation of schedule S' (written $S \leftrightarrow S'$), if$len(S) = len(S')$ and for every $OP_t^{l_i} \in S$, there exists a unique $OP_t^{l_j} \in S'$.

Definition [Serial Schedule.] Schedule $S = OP_{t_1}^{l_1} \ldots OP_{t_n}^{l_n}$ is serial if for all $OP_{t_j}^{l_j}$ there does not exist $OP_{t_k}^{l_j}$, $k > j$ such that $t_k < t_j$.

Lemma *[Permutation.]* Let schedule $S = OP_{t_1}^{l_1}.OP_{t_2}^{l_2}$ be safe. Then if S is safe, there exists a serial schedule S' such that $S \leftrightarrow S'$.

Theorem 5.2. [Deterministic Semantics.] *If schedule S is safe, then there exists a serial schedule S' such that $S \leftrightarrow S'$.*

Theorem 5.3. [Deterministic Semantics.] *Any schedule χ produced by a Pāñini program is safe, and thus Pāñini guarantees deterministic semantics.*

Proof Sketch: Case 1: Suppose neither tasks t or t' write to a common field. That is, without loss of generality, $\nexists loc, f$ s.t. $(\mathtt{wt}\,, t, loc, f) \in \chi \wedge [(\mathtt{wt}\,, t', loc, f) \in \chi' \vee (\mathtt{rd}\,, t', loc, f) \in \chi']$. Then, the first two conditions (*ssafe* and *psafe*) in the previous definition (Schedule Safety) hold.

Case 2: Suppose task t writes a common field with t'. That is, $\exists loc, f$ s.t. $(\mathtt{wt}\,, t, loc, f) \in \chi \wedge [(\mathtt{wt}\,, t', loc, f) \in \chi' \vee (\mathtt{rd}\,, t', loc, f) \in \chi']$. In this case, Pāñini will never schedule t and t' to run concurrently (by the hierarchy built by the (REGISTER) rule and the auxiliary functions it uses – specifically *reorderLvl*). Thus, the first two conditions (*ssafe* and *psafe*) in the previous definition (Schedule Safety) hold.

Finally, the current version of the hybrid system makes register effects conflict with all other effects. Therefore, the last two conditions (*rsafe* and *asafe*) hold. ∎

5.3 Type Soundness

Type Soundness. The proof of soundness of Pāñini's type-and-effect system uses a standard preservation and progress argument [59]. The details are adapted from previous work [16, 25]. Throughout this section we assume a fixed, well-typed program with a fixed class table, CT. A type environment $\Pi ::= \{I : \{t, \rho\}\}$ maps variables and store locations to types and effect sets. The effect set was used in the semantics to compute the dependency between handlers and will not be used in the following section. For simplicity, we omit the effect sets ρ in subsequent discussion. The key definition of consistency is as follows (the auxiliary function if defined in Figure 13).

Definition [Environment-Store Consistency.] Suppose we have a type environment Π and μ a store. Then μ *is consistent with* Π, written $\mu \approx \Pi$, if and only if all the followings hold:

1. $\forall loc \cdot \mu(loc) = [t.F] \Rightarrow$
   (a) $\Pi(loc) = t$ and
   (b) $dom(F) = dom(\mathit{fieldsOf}(t))$ and
   (c) $rng(F) \subseteq dom(\mu) \cup \{\mathtt{null}\}$ and
   (d) $\forall f \in dom(F) \cdot F(f) = loc'$, $\mathit{fieldsOf}(t)(f) = u$ and $\mu(loc') = [t'.F'] \Rightarrow t' <: u$
2. $\forall loc \cdot loc \in dom(\Pi) \Rightarrow loc \in dom(\mu)$

We now state the standard lemmas for substitution, extension, environment contraction, replacement and replacement with subtyping. These lemmas can be proved by adaptations of Clifton's proofs for MiniMAO_0 [16].

Lemma *[Substitution.]* If $\Pi, var_1 : t_1, \ldots, var_n : t_n \vdash e : t$ and $\forall i \in \{1..n\} \cdot \Pi \vdash e_i : s_i$ where $s_i <: t_i$ then $\Pi \vdash [var_1/e_1, \ldots, var_n/e_n]e : s$ for some $s <: t$.

Proof Sketch: The proof proceeds by structural induction on the derivation of $\Pi \vdash e : t$ and by cases based on the last step in that derivation. The base cases are (T-NEW), (T-NULL) and (T-VAR), which have no variables and $s = t$. Other cases can be proved by adaptations of MiniMAO_0 [16]. The induction hypothesis (IH) is that the lemma holds for all sub-derivations of the derivation. The cases for (T-CAST), (T-SEQUENCE), (T-SET), (T-CALL) and (T-GET) are similar to Clifton's proofs. We now consider the case for (T-DEFINE), (T-REGISTER), (T-ANNOUNCE) and (T-YIELD).

For $c\ var = e_1; e_2$, by IH, if we substitute the variables for e_1, we will get the subtype of e_1, which is a subtype of c. Also, the type for the substitution for e_2 results in a subtype of it. Therefore, since the type for the entire expression is the type for e_2, it holds.

For **announce** $p\,(e_1, \ldots, e_n)$, we do the same substitution for each argument $e_i, 1 \leq i \leq n$. By IH, each of these has a subtype of the argument. Therefore, since the whole expression has the type **void**, consistency holds.

The cases for **yield** e and **register**(e) are straightforward, because the type of **yield** e and **register**(e) is the same as e. ∎

Lemma *[Environment Extension.]* If $\Pi \vdash e : t$ and $a \notin dom(\Pi)$, then $\Pi, a : t' \vdash e : t$.

Proof Sketch: The proof is by a straightforward structural induction on the derivation of $\Pi \vdash e : t$. The base cases are (T-NEW), (T-NULL) and (T-VAR). In the first two cases,

the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the (T-VAR) case, $e = var$ and $\Pi(var) = t$. But $a \notin dom(\Pi)$, so $var \neq a$. Therefore $(\Pi, a : t')(var) = t$ and the claim holds for this case. The remaining typing rules cover the induction step. By the induction hypothesis, changing the type environment to $\Pi, a : t'$ does not change the types assigned by any hypotheses. Therefore, the types assigned by each rule are also unchanged and the claim holds. ∎

Lemma *[Environment Contraction.]* If $\Pi, a : t' \vdash e : t$ and a is not free in e, then $\Pi \vdash e : t$.

Proof Sketch: The proof is by a straightforward structural induction on the derivation of $\Pi, a : t' \vdash e : t$. The base cases are (T-NEW), (T-NULL) and (T-VAR). In the first two cases, the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the (T-VAR) case, $e = var$ and $(\Pi, a : t')(var) = t$. But a is not free in e, so $var \neq a$. Therefore $\Pi(var) = t$ and the claim holds for this case. The remaining typing rules cover the induction step. By the induction hypothesis, changing the type environment to Π does not change the types assigned by any hypotheses. Therefore, the types assigned by each rule are also unchanged and the claim holds. ∎

Lemma *[Replacement.]* If $\Pi \vdash \mathbb{E}[e] : t, \Pi \vdash e : t'$, and $\Pi \vdash e' : t'$, then $\Pi \vdash \mathbb{E}[e'] : t$.

Proof Sketch: By examining the evaluation context rules and corresponding typing rules, we see that $\Pi \vdash e : t'$ be a sub-derivation of $\Pi \vdash \mathbb{E}[e] : t$. Now the typing derivation for $\Pi \vdash \mathbb{E}[e'] : t''$ must have the same shape as that for $\mathbb{E}[e] : t$, except for the sub-derivation for $\Pi \vdash e' : t'$. However, because this sub-derivation yields the same type as the sub-derivation it replaces, it must be the case that $t'' = t$. ∎

Lemma *[Replacement with Subtyping.]* If $\Pi \vdash \mathbb{E}[e] : t, \Pi \vdash e : u$, and $\Pi \vdash e' : u'$ where $u' <: u$, then $\Pi \vdash \mathbb{E}[e'] : t'$ where $t' <: t$.

Proof Sketch: The proof is by induction on the size of the evaluation context $\mathbb{E}$, where the size is the number of recursive applications of the syntactic rules necessary to build $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $t' = u' <: u = t$. For the induction step we divide the evaluation context into two parts so that $\mathbb{E}[-] = \mathbb{E}_1[\mathbb{E}_2[-]]$, where $\mathbb{E}_2$ has size one. The induction hypothesis is that the claim of the lemma holds for all evaluation contexts smaller than the one considered in the induction step. We use a case analysis on the rule used to generate $\mathbb{E}_2$. In each case we show that $\Pi \vdash \mathbb{E}_2[e] : s$ implies that $\Pi \vdash \mathbb{E}_2[e'] : s'$, for some $s <: s$, and therefore the claim holds by the induction hypothesis. The cases for (T-CAST), (T-SEQUENCE), (T-SET), (T-CALL) and (T-GET) are similar to Clifton's proofs. We now consider the case for (T-DEFINE), (T-REGISTER), (T-ANNOUNCE) and (T-YIELD).

(a) Cases $\mathbb{E}_2 = -; e_2$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be (T-DEFINE):

$$\frac{\textit{isClass}(c) \qquad \Pi \vdash e : u \qquad \Pi, var : c \vdash e_2 : s \qquad u <: c}{\Pi \vdash \mathbb{E}[e] : s}$$

Now, $u' <: u <: c$, so by (T-DEFINE), $\Pi \vdash \mathbb{E}[e'] : s$.

(b) Cases $\mathbb{E}_2 = \texttt{announce}\ (v_1, \ldots, v_{p-1}, -, e_{p+1}, \ldots, e_n)$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be (T-ANNOUNCE):

$$\frac{\begin{array}{c} CT(p) = \texttt{event}\ p\ \{t_1\ var_1\texttt{;}\ \ldots\ t_n\ var_n\texttt{;}\ \} \\ (\forall\, i \in \{1..(p-1)\}\ ::\ \Pi \vdash v_i : t'_i\ \wedge\ t'_i <: t_i) \qquad \Pi \vdash e : u \\ u <: t_p \qquad (\forall\, j \in \{(p+1)..n\}\ ::\ \Pi \vdash e_j : t'_j\ \wedge\ t'_j <: t_j) \end{array}}{\Pi \vdash \mathbb{E}[e] : \texttt{void}}$$

Now, $u' <: u <: s_p$, so by (T-ANNOUNCE), $\Pi \vdash \mathbb{E}[e'] :$ **void**.

(c) Cases $\mathbb{E}_2 = \texttt{register}(-)$. The last step for $\mathbb{E}_2[e]$ must be (T-REGISTER):

$$\frac{\Pi \vdash e : t \qquad \textit{isClass}(t)}{\Pi \vdash \mathbb{E}[e] : t}$$

Now, $t' <: t$, so by (T-REGISTER), $\Pi \vdash \mathbb{E}[e'] : t' <: t$.

(d) Cases $\mathbb{E}_2 = \texttt{yield}\ (-)$. The last step for $\mathbb{E}_2[e]$ must be (T-YIELD):

$$\frac{\Pi \vdash e : t}{\Pi \vdash \mathbb{E}[e] : t}$$

Now, $t' <: t$, so by (T-YIELD), $\Pi \vdash \mathbb{E}[e'] : t' <: t$. ∎

Theorem 5.4. [Progress.] *For a well-typed expression e, a task dependencies τ, a task queue ψ, a store μ, and a handler queue γ. If $\Pi \vdash e : t$ and $\mu \approx \Pi$, then either $e = loc$ or $e =$ **null** or $e =$ `NullPointerException` or $e =$ `ClassCastException` or $\langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle e', \tau'\rangle + \psi', \mu', \gamma'\rangle$.*

Proof Sketch:

(a) If $e = v$ or $e =$ **null**, it is trivial.

(b) Cases $e =$ `NullPointerException` or $e =$ `ClassCastException`, which are final states of the programs, result from the semantics rules **null**.f, **null**.$f = v$, **null**.$m(v_1, \ldots, v_n)$, **register**(**null**) and **cast** e. We presented the rules in Figure 21. These values serve as the base cases.

(NCALL)
$$\langle\langle \mathbb{E}[\mathtt{null}.m(v_1,\ldots,v_n)],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle NullPointerException,\tau\rangle,\mu,\gamma\rangle$$

(NGET)
$$\langle\langle \mathbb{E}[\mathtt{null}.f],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle NullPointerException,\tau\rangle,\mu,\gamma\rangle$$

(NSET)
$$\langle\langle \mathbb{E}[\mathtt{null}.f = v],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle NullPointerException,\tau\rangle,\mu',\gamma\rangle$$

(XCAST)
$$\frac{[c'.F] = \mu(loc) \qquad c' \not<: c}{\langle\langle \mathbb{E}[\mathtt{cast}\ c\ loc],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle ClassCastException,\tau\rangle,\mu,\gamma\rangle}$$

Figure 21. Operational semantics of expressions that produce exceptions, base on,[46].

(c) In the case where the expression e is not a value, evaluation rules are considered case by case for the proof. We proceed with the induction of derivation of expression e. Induction hypothesis (IH) assumes that all sub-terms of e progress and are well-typed.

Cases $e = \mathbb{E}[\mathbf{new}\ c()]$, $e = \mathbb{E}[loc.m(v_1,\ldots,v_n)]$, $e = \mathbb{E}[loc.f]$, $e = \mathbb{E}[loc.f = v]$, $e = \mathbb{E}[\mathbf{cast}\ t\ loc]$, $e = \mathbb{E}[t\ var = v; e]$ and $e = \mathbb{E}[v; e_1]$ are similar to Clifton's work [16] and are omitted.

Case $e = \mathbb{E}[\mathbf{register}\ e]$. Based on the IH, e is well typed. Thus, it evolves by (RE-REGISTER) or (REGISTER).

Case $e = \mathbb{E}[\mathbf{announce}\ p\ (v_1,\ldots,v_n)]$. Based on the IH, p is well typed and is defined. Each parameter is well typed and is a subtype of the type of the field in event p. Thus, it evolves by (ANNOUNCE).

Case $e = \mathbb{E}[\mathbf{yield}\ e]$. This case has no constraint and evolves based on different rules. ∎

Theorem 5.5. [Subject-reduction.] *Let e be an expression and $e \neq \mathbf{yield}\ e_1$ for any e_1, τ task dependencies, ψ a task queue, μ a store, and γ a handler queue. Let Π be a type environment such that $\mu \approx \Pi$. And let t a type. If $\Pi \vdash e : t$ and $\langle\langle e,\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle e',\tau'\rangle + \psi',\mu',\gamma'\rangle$, then there is some $\mu' \approx \Pi'$ and t' such that $\Pi' \vdash e' : t'$ and $t' <: t$.*

Proof Sketch: The proof is by cases on the definition of $\hookrightarrow$ separately. The cases for object oriented parts (rules (NEW), (NULL), (CAST), (GET), (SET), (VAR), (SEQUENCE) and (CALL)) can be proved by adaptations of Clifton's proofs for MiniMAO_0 [16].

The rule for (SEQUENCE) is similar to Clifton's work, except that $e' = \mathbb{E}[\mathbf{yield}\ e]$ instead of $e' = \mathbb{E}[e]$. Since the type of $\mathbf{yield}\ e$ has the same type as e, this case holds. For (DEFINE), $e = \mathbb{E}[t\ var = v; e_1]$ and $e' = \mathbb{E}[[var/v]e_1]$: let $\tau' = \tau$, $\mu' = \mu$, $\psi' = \psi$, $\gamma' = \gamma$ and $\Pi' = \Pi$. We now show that $\Pi \vdash e' : t'$ for some $t' <: t$. $\Pi \vdash e : t$ implies that $t\ var = v; e_1$ and all its subterms are well typed in Π. Let $\Pi \vdash (t\ var = v; e_1) : u$. By (`T-Define`), $\Pi, var : t \vdash e_1 : u'$. By Lemma 5.3, $\Pi \vdash [var/v]e_1 : u''$ for some $u'' <: u' <: u$. Therefore, by lemma 5.3, $\Pi \vdash e' : t'$ for some $t' <: t$. For the (RE-REGISTER) rule, $e = \mathbb{E}[\mathbf{register}(v)]$ and $e' = \mathbb{E}[v]$. Let $\tau' = \tau$, $\mu' = \mu$, $\psi' = \psi$, $\gamma' = \gamma$ and $\Pi' = \Pi$. Obviously, $t' = t$. For the (REGISTER) rule, $e = \mathbb{E}[\mathbf{register}(v)]$ and $e' = \mathbb{E}[v]$. Let $\tau' = \tau$, $\mu' = \mu$, $\psi' = \psi$, $\gamma' = v + \gamma$ and $\Pi' = \Pi$. Clearly, $t' = t$. For the (ANNOUNCE) rule, $e' = \mathbb{E}[e_2]$ and $e = \mathbb{E}[\mathbf{announce}\ p\ \{v_1,\ldots,v_n\}]$. Let $\mu' = \mu$, $\gamma' = \gamma$, $\Pi' = \Pi$ and $t' = t$. $\mathbf{void} <: c$ for any class type c. ∎

Definition [Thread-interleaving.]
If $\langle\langle \mathbb{E}[\mathbf{yield}\ e],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle \mathbb{E}_1[e_1],\tau_1\rangle + \psi_1,\mu_1,\gamma_1\rangle \ldots \hookrightarrow \langle\langle \mathbb{E}_n[e_n],\tau_n\rangle + \psi_n,\mu_n,\gamma_n\rangle \hookrightarrow \langle\langle \mathbb{E}[e],\tau\rangle + \psi',\mu',\gamma'\rangle$

or $\langle\langle \mathbb{E}[\mathbf{yield}\ e],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[e],\tau\rangle + \psi',\mu',\gamma'\rangle$,
we denote this as $\langle\langle \mathbb{E}[\mathbf{yield}\ e],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow^* \langle\langle \mathbb{E}[e],\tau\rangle + \psi',\mu',\gamma'\rangle$, where $\forall i\{1 \le i \le n\}\ \mathbb{E}_i[e_i] \neq \mathbb{E}[e]$.

Theorem 5.6. [Subject-reduction-Thread-interleaving.] *For an expression $e = \mathbb{E}[\mathbf{yield}\ e_1]$, for any e_1, τ task dependencies, and ψ a task queue, μ a store and γ a handler queue. Let Π be a type environment such that $\mu \approx \Pi$. And let t a type. If $\Pi \vdash \mathbb{E}[\mathbf{yield}\ e_1] : t$ and $\langle\langle \mathbb{E}[\mathbf{yield}\ e_1],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow^* \langle\langle \mathbb{E}[e_1],\tau'\rangle + \psi',\mu',\gamma'\rangle$, then there is some $\mu' \approx \Pi'$ and t' such that $\Pi' \vdash \mathbb{E}[e_1] : t'$ and $t' <: t$.*

Proof Sketch: The proof is by induction on the number n of **yield** expressions in the transitions.

In the base case, $n = 0$, $\langle\langle \mathbb{E}[\mathbf{yield}\ e],\tau\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[e],\tau\rangle + \psi,\mu,\gamma\rangle$. Let $\Pi' = \Pi$ and $t' = t$. The condition holds.

If $n = 1$, $\langle\langle \mathbb{E}[\mathbf{yield}\ e],\langle t, I\rangle\rangle + \psi,\mu,\gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[e_1],\tau_1\rangle + \psi_1,\mu,\gamma\rangle \hookrightarrow^* \langle\langle \mathbb{E}[\mathbf{yield}\ e_1'],\tau_1\rangle + \psi_1',\mu_1',\gamma_1'\rangle \hookrightarrow \langle\langle \mathbb{E}[e],\tau\rangle + \psi_1',\mu_1',\gamma_1'\rangle$. And $\Pi \vdash t$. Since $\mu \approx \Pi$, $\exists t_1 :: \Pi \vdash \mathbb{E}[e_1] : t_1$. By Theorem 5.5, $\exists t_1', \Pi_1 :: \Pi_1 \vdash \mathbb{E}[\mathbf{yield}\ e_1'] : t_1' \wedge \mu_1' \approx \Pi_1$. Therefore, $\Pi_1 \vdash \mathbb{E}[e] : t' \wedge t' <: t$.

The (IH) is that there is some $\mu' \approx \Pi'$ and t' such that $\Pi' \vdash \mathbb{E}[e_1] : t'$ and $t' <: t$ for $\forall i :: 1 \le i \le n$, the number of transitions. By Theorem 5.5 and it also true for the last transition, the claim is also true, by adding one more transition. ∎

Theorem 5.7. [Soundness.] *Given a program $P = \mathbf{decl}_1 \ldots \mathbf{decl}_n\ e$, if $\vdash P : (t,\rho)$ for some t and ρ, then either the evaluation of e diverges or else $\langle\langle e,\langle 0,\emptyset\rangle\rangle,\bullet,\bullet\rangle \hookrightarrow^* \langle\langle v,\tau'\rangle,\mu',\gamma'\rangle$ where one of the following holds for v: $v = loc$ or $v = \mathbf{null}$ or $v = \mathtt{NullPointerException}$ or $v = \mathtt{ClassCastException}$.*

Proof Sketch: If e diverges, then this case is trivial. Otherwise if e converges, then because the empty environment is consistent with the empty store. This case is proved by Theorem 5.4, Theorem 5.5 and Theorem 5.6. ∎

6. Evaluation: Performance Benefits

We now evaluate the performance benefits of our type-and-effect system using several real world applications. To facilitate these experiments we enhanced the compiler for the Pāñini language [36] to incorporate our type-and-effect system. All performance-related experiments were run on a system with a total of 24 cores (two 12-core AMD Opteron 6168 chips 1.9GHz) running Fedora GNU/Linux. In the experiments, the number of "threads" is often varied. Pāñini's implementation uses a thread pool of n threads. So, using n threads is essentially the same as enabling n cores (when $n \leq 24$ for our system). For each of the experiments, an average of the results over ten runs was taken.

6.1 Candidate Java Applications

We have studied several Java applications. Figure 22 presents key static characteristics of these applications as well as the speedup seen in each case. In this table, the column labeled *lines* shows the lines of source code in the application (not including comments and blank lines). Columns labeled *methods* and *classes* show the total number of Java methods and classes respectively. Finally, the column labeled *Speedup* shows the maximum observed speedup.

Application	Size			Speedup
	lines	methods	classes	
FindBugs [31]	110932	21508	2781	5.7x
jASEN [2]	11497	955	165	1.7x
RefactoringCrawler [17]	7862	914	268	3.5x
WebSPHINX [41]	16061	1807	216	15.7x
Genetic Algorithm [48]	461	121	20	7.3x

Figure 22. Static characteristics of evaluation candidates. "lines" is counted as non-comment, non-blank LOC [3].

In the following sections, we first introduce each application, describe the applicability of our hybrid type-and-effect system to that application, and describe performance benefits observed for that application.

6.2 Candidate: FindBugs — Static Bug Detection

First we present performance results for FindBugs, a static analysis tool for bug detection, described in Section 1.

6.2.1 Experimental Setup and Results

Our implementation is similar to the example presented in Figure 3. In our implementation, we registered 7 bug detectors that may all run safely in parallel.

Figure 23 shows the speedup for FindBugs with varying number of threads in bug analysis code.

This figure shows the speedup of FindBugs implemented in Pāñini as well as a manually parallelized version. In the manually parallelized code, we created 7 tasks, which respectively wrap those 7 detectors as in the Pāñini code. These tasks are then submitted to the fork/join framework [33] for execution[3]. Once the task is taken by the framework for execution, the `check` method of the wrapped detector will be invoked. As the figure shows, the speedups between the manually parallelized version and the Pāñini version are comparable. Both implementations achieved nearly 6x speedup with 7 or more threads.

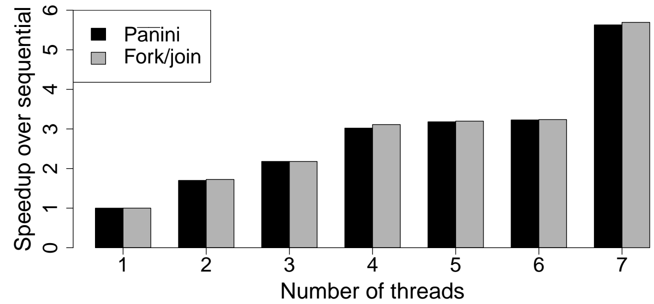

Figure 23. FindBugs: speedup over sequential version. Pāñini achieves nearly 6x speedup for 7 threads. Speedup is nearly the same as manually parallelized version suggesting negligible overhead.

6.2.2 Analysis of results

An interesting result shown in this figure is the plateau from 4 to 6 threads and the sudden increase in speedup when increasing the number of threads from 6 to 7. To understand this, first consider Figure 24 which shows the portion of sequential runtime for each of the 7 detectors.

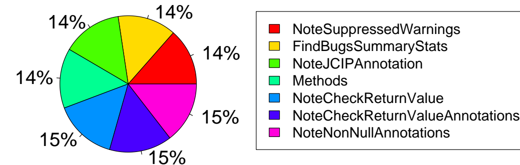

Figure 24. FindBugs: breakup of sequential runtime by detectors, shows that detectors take similar time to finish.

As this figure shows, all of these 7 detectors take roughly the same amount of time to execute. Therefore, with 6 threads, the first 6 detectors will execute concurrently and the 7th detector can not be invoked until one of the first 6 detectors finishes. Thus, it will take approximately as long as two detectors running sequentially. A similar situation occurs for 4 and 5 threads.

6.3 Candidate: jASEN — Anti Spam ENgine

jASEN is a Java anti spam engine combining bayesian-like scanning with intelligent email inspection and classification [2]. This tool is best suited to developers wishing to integrate anti-spam services into an existing server based Java email application. jASEN allows adding/removing different spam detection algorithms (called *plugins* in its implementation). Figure 25 shows the main logic from jASEN.

The class `Jasen` is the core scanning class of the jASEN framework. It scans a message using any registered plug-

[3] We use the fork/join framework because of its efficiency.

```
1 class Jasen {
2   Result scan(Message m) {
3     // initilization code omitted
4     Result r;
5     for(Plugin p:plugins) {
6       r = p.test(m);
7       if(r.isAbsolute()) {
8         return r;
9       }
10      // further processing omitted
11    }
12  }
13 }
```

```
14 class AttachScanner implements Plugin {
15   Result test(Message m) {
16     PointResult r = /*...*/;
17     if(/*...*/) {
18       r.setAbsolute(true);
19     }
20     return r;
21   }
22 }
```

```
23 class Keyword implements Plugin {
24   Result test(Message m) {
25     ProbabilityResult r = /*...*/;
26     if(/*...*/) {
27       r.setProbability(highProb);
28     }
29     return r;
30   }
31 }
```

Figure 25. Snippets from jASEN, an Anti Spam ENgine. Two concrete `Plugins` (handlers) are shown.

```
1 class Jasen {
2   Result scan(Message m) {
3     // initilization code omitted
4     announce messageAvailable(m);
5     // subsequent code unchanged
6   }
7 }

9 class Keyword extends Plugin{
10  void test(Message m) {
11    if(done){
12      return;
13    }
14    // subsequent code unchanged
15  }
16 }
```

```
17 event messageAvailable {
18   Message m;
19 }

21 class Plugin {
22   when messageAvailable do test;
23   when resultFound do finish;
24   boolean done = false;
25   void init(){ register(this); }
26   void finish(Result result){
27     done = true;
28   }
29   void test(){}
30 }
```

```
31 event resultFound {
32   Result result;
33 }

35 class AttachScanner extends Plugin {
36   void test(Message m) {
37     if(done){
38       return;
39     }
40     PointTestResult result = /*...*/;
41     if(/*...*/) {
42       result.setAbsolute(true);
43       announce resultFound(result);
44     }
45   }
46 }
```

Figure 26. Pāñini implementation of jASEN. Imperative code for implicit invocation is replaced by language features.

ins and returns a scan result object indicating the results of the scan. After initializing the received message, declared on line 2, the method `scan` invokes the registered plugins to detect spam (line 5). Some of the plugins can make an absolute decision and classify an email as a spam. For example, the scanner `AttachScanner` in some cases can tag a message as spam (line 18). Other plugins can only set a certain value for the probability of a message being a spam. For example, the `Keyword` class, may only set a probability of a message being a spam (line 27). We refer to these detectors as absolute and probability detectors respectively in subsequent discussion. Once an e-mail is detected to be spam, the method `scan` will return it immediately (line 7).

This plugin-invocation logic is implemented as the observer pattern [26] with the method `scan` as the subject and plugins as handlers. A Pāñini version is shown in Figure 26.

For the Pāñini version, the code for the method `scan` is almost the same. The main difference is that an event announcement statement is used to notify the plugin's handlers (line 4) in order to decouple this method from the plugins. Since the basic structure of the plugin's handlers are similar, except for the actual detection logic, this structure is abstracted into a superclass `Plugin`. Concrete plugin detectors extend this class (e.g. `AttachScanner` and `Keyword`). Once an event of type `messageAvailable` is fired, the method `test` will be invoked. The logic for setting an e-mail as spam and ending the detection is implemented via introducing another event of type `resultFound`. As before, the `AttachScanner` plugin can classify a message as spam and, in this case, announces an event of type `resultFound` (line 43). Once a message is tagged as an spam, the plugins handlers are notified (line 23), an instance field `done` (line 27) in the superclass is set, and the actual class will not process any longer (line 37 and line 11).

To enhance the scalability of this spam filter, it is desirable to execute the non-conflicting plugins concurrently. However, as shown in Figure 26, on line 27, the plugin can classify a spam and stop others. In this case, it announces an event of type `resultFound`, and plugin handlers will accept this event, (line 23) and write to instance fields `done` (line 27). Therefore, the effects of this absolute plugin are {**write** `Plugin done`, **read** `Plugin done`}. Probability plugins will read their corresponding instance field `done`, e.g. on line 11, and the effect is {**read** `Plugin done`}. Because of this, the absolute detector conflicts with all other detectors (note that probability detectors do an instance field read and do not conflict with other probability detectors). What is more, since the method `scan` is unaware of whether these absolute detectors will register or not, nor does it statically know the registration order of the plugins, a static type-and-effect system will conservatively sequentialize the entire detection [45]. However, our type-and-effect system is able to organize the execution schedule to maximize concurrency and eliminate unsafe executions. It will execute the absolute plugins sequentially, while execute the probability plugins in parallel. Notice that this schedule is

sequentially consistent [9] with the sequential execution, i.e. the results they produce are the same.

6.3.1 Experimental Setup and Results

In this experiment, we registered 14 probability plugins and measured the speedup. The results are shown in Figure 27.

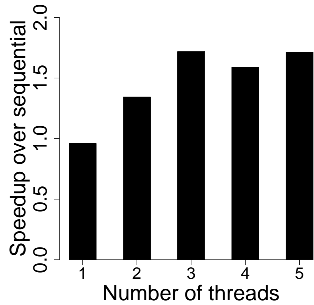

Figure 27. jASEN: speedup over sequential version

The figure indicates that maximum speedup was around 1.7x speedup in spam detection code.

6.3.2 Analysis of results

To understand these speedups, consider Figure 28 which shows the proportion of sequential time taken for each of the 14 detectors.

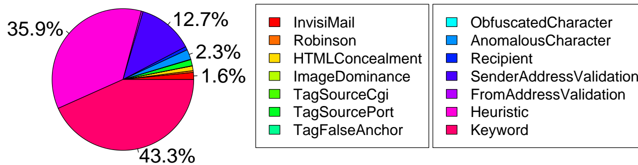

Figure 28. jASEN: breakup of sequential runtime by plug-ins, shows that 3 plugins dominate the execution time.

As the figure shows, three of detectors make up most of the execution time. As a result of this, the speedups level off when more than 3 threads are available.

6.4 Candidate: Refactoring Crawler

Refactoring Crawler is a tool for detecting refactoring between software versions [17]. This tool is useful for updating software to use the latest version of its libraries, which are constantly changing. Currently, the tool detects refactorings like renaming package, class and method, pullup and push-down method, move method and changes of method signatures. A code snippet is presented in Figure 29.

The class `DetectRefactoringsPlugin` is the driver class which initializes the *reference graphs*, as well as other variables for the detection. After initialization, it invokes different detectors, on lines 5-11, to crawl refactorings between two versions of a Java application. These detectors may or may not be invoked for a given run based on the user input. This behavior can be seen at the **`if`** conditions on line 5.

```
1 class DetectRefactoringsPlugin {
2   void doLaunch(/**/) {
3     /* The initializations, e.g. the reference
4     graphs for both versions, are omitted. */
5     if (/*...*/) {
6       detectPullUpMethod(/*...*/);
7     }
8     /* other detectors omitted */
9     if (/*...*/){
10      detectMoveMethod(/*...*/);
11    }
12  }
13 }
```

Figure 29. Snippets from Refactoring Crawler. Function `doLaunch` will call enabled detectors.

```
1 event detect{
2   Graph originalGraph;
3   Graph versionGraph;
4 }

6 class DetectRefactoringsPlugin {
7   void doLaunch(/**/) {
8   /* the initilizations code remains. */
9     announce detectRefactoring(
10      originalGraph, versionGraph);
11  }
12 }

14 class RenameMethodDetection{
15  when detectRefactoring do detect;
16  void init(){ register(this); }
17  void detect() {
18    detectRenameMethod(/*...*/);
19  }
20 }

22 class MoveMethodDetection {
23  when detectRefactoring do detect;
24  void init(){ register(this); }
25  void detect() {
26 /* the actually detection algorithm is omitted. */
27    if (/*PulledUpCategory.contain(prunedPair)
28    || PushedDownCategory.contain(prunedPair)*/) {
29      pairsToRemove.add(prunedPair);
30    }
31  }
32 }
```

Figure 30. Pāñini version of the Refactoring Crawler. Imperative code for implicit invocation is replaced by language features.

Again, the observer pattern [26] is used to decouple `doLaunch` and refactoring detectors which are the subject and handlers respectively. The Pāñini version of the code is shown in Figure 30.

In the Pāñini version of the code, the initializations in the `doLaunch` method remain unchanged. Once the initializations are done, this method signals an event of type `detect` and the detectors are notified on line 9.

All of these detectors are computationally intensive. Thus to enhance the scalability of this application, it is desirable to execute the detectors concurrently. However, as is shown in Figure 30 the `MoveMethodDetection` detector read instance fields (line 27), which could be written by the `PulledUpMethodDetection` and `PushDownMethod` detectors. It reads the fields to check whether a certain po-

tential refactoring is already classified as another refactoring. As a result of this, the detector `MoveMethodDetection` conflicts with these other two detectors (note that other detectors do not conflict with each other). What is more, since the method `doLaunch` do not statically know whether this detector will register or not, neither does it know in advance about registration order of these detectors, a static type-and-effect system will conservatively sequentialize the entire detection process [45]. However, our hybrid type-and-effect system is able to manage the handler list and organize the execution schedule to maximize concurrency and eliminate unsafe executions. For example, if all the 7 handlers register in the order described in the first paragraph of this section, it will execute the other 6 handlers in parallel and execute the `MoveMethodDetection` detector after the other handlers are done.

6.4.1 Experimental Setup and Results

In the first experiment, we registered all 7 detectors in the order in Figure 29. Figure 31 shows the speedup with varying number of threads.

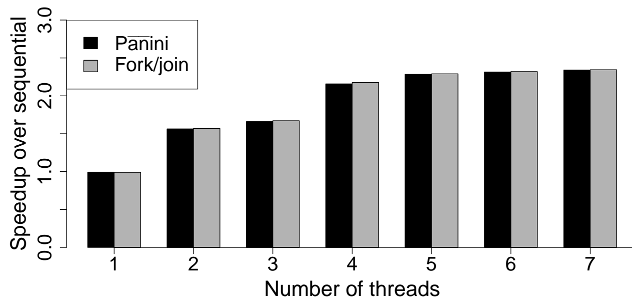

Figure 31. Refactoring Crawler: speedup over sequential version.

This figure shows the speedup of the Pāṇini version as well as a manually parallelized version. The manually parallelized version is created in a manner similar to that of FindBugs as described in Section 6.2.1.

As the figure shows, speedups between the manual version and the Pāṇini versions are comparable. The speedups achieved are almost 2.5x with 5-7 threads in the detection code. To understand why speedup does not go beyond 2.5x, first consider Figure 32.

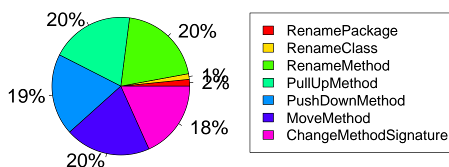

Figure 32. Refactoring Crawler: breakup of sequential runtime by refactoring detectors

Figure 32 shows the portion of sequential execution time for each of these 7 detectors. As we can see, the detectors for rename package and class, respectively, take far less time to execute. That is expected, because generally, there are far more methods in a Java application than classes or packages. All the other 5 detectors take roughly the same amount of time for execution. Notice that as mentioned in the previous section, the moved method detector could only be executed after all other detectors are done because it conflicts with other detectors. This explains the large jump from 1 thread to 2 threads and from 3 threads to 4 threads. Finally, speedup may not be more than 3 because the moved method detector conflicts with other refactoring detectors.

In the next experiment, we did not register the move method detector. Thus, all 6 remaining methods may run in parallel. These results are shown in Figure 33.

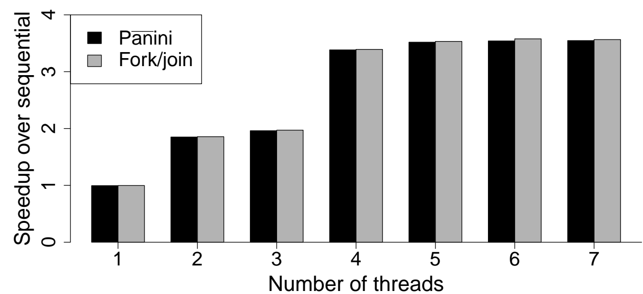

Figure 33. Refactoring Crawler: speedup over sequential version, without the conflicting handler.

The results are expected in that it achieved almost 4x speedup. This is because there are 4 detectors, each may safely run in parallel, and each has a comparable runtime.

6.5 Candidate: WebSPHINX — Crawling the WWW

WebSPHINX is a web crawler for collecting links [41]. Figure 34 shows the main logic from this application.

```
1 void process (Link link) {
2   Page page = link.getPage ();
3   for(Classifier c: classifiers) {
4     c.classify(page);
5   }
6   expand (page);
7   // processing detail omitted
8 }

10 void expand (Page page) {
11   if(page.depth() >= max) return;
12   for(Link l: page.links())
13     process(ls[i]);
14 }
```

Figure 34. Snippets from WebSPHINX, a web crawler. Iteratively calls each classifier on each page.

Upon receiving a link, this crawler fetches the corresponding page. It invokes classifiers to annotate pages (lines 3-5). It expands the crawl from this page by calling

the `expand` method. It processes all the links referenced to by the page until certain criteria are met, e.g. a certain depth is reached. After expanding the page (line 6), it continues processing the page (not shown). Again, the observer pattern [26] is used to decouple `Classifiers` from the `process` method.

```
1 event PageAvailable{
2   Page page;
3 }

5 void expand (Page page) {
6   if(page.depth() >= max) return;
7   List<link> ls = page.links();
8   if(!ls.isEmpty()){
9     announce LinkAvailable(ls);
10  }
11 }

13 event LinksAvailable{
14   List ls;
15 }
16 class processLink {
17   when LinkAvailable do process;
18   void init(){ register(this); }
19   void process(List links) {
20     process(cast Link links.get(0));
21   }
22 }

24 void process (Link link) {
25   Page page = link.getPage ();
26   announce PageAvailable(page);
27   expand (page);
28   // processing detail omitted
29 }
30 class processRest {
31   when LinkAvailable do process;
32   void init(){ register(this); }
33   void process(List links) {
34     List ls = new ArrayList(links);
35     ls.remove(0);
36     if(!ls.isEmpty()){
37       announce LinkAvailable(ls);
38     }
39   }
40 }
```

Figure 35. Pāñini version of WebSPHINX, web crawler. Imperative code for implicit invocation is replaced by language features.

It is beneficial to expose potential concurrency in processing links on lines 12-13. Several, but not all, concrete classes that implement this `Classifier` interface write to the same location (they are also not commutative operations [47]). However, it is beneficial to parallelize the methods that use the default classifier (not shown) which does not write to a same location, as well as other cases that use other non-conflicting classifiers.

Snippets from the Pāñini version of the web crawler code are shown in Figure 35.

Like the earlier examples, the implementation is similar to the OO version except that the for loop is replaced by an event announcement statement. We register a continuation handler, shown on line 30, and a link processor, on line 16, for this event. Thus traversing the list of classifiers is now delegated to the event system on line 26.

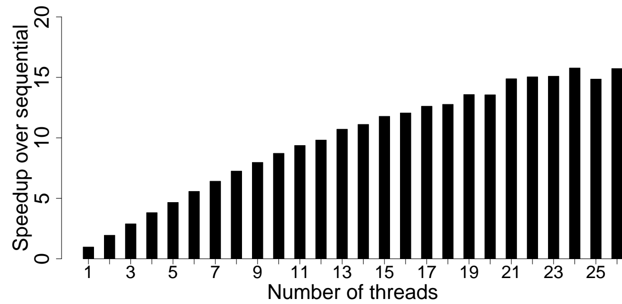

Figure 36. Web crawler: speedup over sequential. Speedup scales well (up to almost 16x) until all cores are utilized.

6.5.1 Experimental Setup and Results

For this experiment, we measured the speedup of the Pāñini version of the web crawler over the sequential version. We measured the speedup for web-crawling with a maximum depth of 6. The results are shown in Figure 39. As expected, as the number of threads increases, so does speedup. It achieved more than 15.7x speedup.

6.6 Candidate: GA — Mutate and Evolve

A genetic algorithm (GA) mimics the process of natural selection. These algorithms are computationally intensive and are useful for solving optimization problems [48]. The main idea is that searching for a desirable state is done by combining two *parent* states, instead of modifying a single state. An initial *generation* with n members is given to the algorithm. A *cross over* function is used to combine different members of the generation to develop the next generation. Optionally, members of the offspring may be randomly *mutated* slightly. Finally, members of the generations (or an entire generation) are ranked using a *fitness function*.

Figure 37 shows two main sub-algorithms of the GA that change a generation to produce a new generation: `CrossOver` and `Mutation`. To allow adding and removing other components in the flow of generations, this algorithm is implemented using the observer design pattern. These sub-algorithms serve as handlers. Once a new generation is produced, these handlers will be notified. In Figure 38, some other handlers are presented, as in JGAP [39]. A monitor will terminate the entire computation once certain criteria are met. A logger could log all the generations produced. Different fitness functions could be used for different purposes. All these other handlers are not needed all the time and may be registered in any combination to handle intermediate outputs. Therefore, it makes sense to decouple them from the main computations [26, pp.293].

Generally, both the mutation and the crossover functions are computationally intensive and have no dependency on each other. Thus, executing these two functions in parallel is beneficial. The effects of both these han-

```
1 event GenReady {  GenCont gct;  }

3 class GenCont{
4   Generation g;
5   boolean done;
6   GenCont(Generation g, boolean done){
7     this.g = g; this.done = done
8   }
9 }

11 class CrossOver {
12   int prob; int max;
13   void init(){ register(this); }
14   when GenReady do cross;
15   void cross(GenCont gct){
16     Generation g = gct.gen();
17     int gSize = g.size();
18     Generation g1 = new Generation(g);
19     // apply crossover funtion on g1;
20     if(g1.getDepth() < max && gct.done)
21       announce GenReady(new GenCont(g1,false));
22   }
23 }

25 class Mutation {
26   int prob; int max;
27   void init(){ register(this); }
28   when GenReady do mutate;
29   void mutate(GenCont gct){
30     Generation g = gct.gen();
31     int gSize = g.size();
32     Generation g1 = new Generation(g);
33     // apply Mutation funtion on g1;
34     if(g1.getDepth() < max && gct.done)
35       announce GenReady(new GenCont(g1,false));
36   }
37 }
```

Figure 37. GA implementation in Pāñini. Cross-over and mutation handlers do not have conflicts.

```
1 class ImprovementMonitor {
2   Generation lastG;
3   void init(){ register(this); }
4   when GenReady do improve;
5   void improve(GenCont gct){
6     Generation g = gct.gen();
7     if(gain(lastG, g)){
8       gct.done = true;
9     }
10    lastG = g;
11  }
12  boolean gain(Generation g, Generation g1) {
13    // pure function that computes value gained.
14  }
15 }

17 class Logger {
18   when GenReady do log;
19   void init(){ register(this); }
20   void log(GenCont gct){
21     logGen(gct.g);
22   }
23 }

25 class OffsetRemoverFitness {
26   Int preOffset;
27   void init(){ register(this); }
28   when GenReady do improve;
29   void evaluate(GenCont gct){
30     Int cur;
31     // Computes fitness value and cur, detail omitted
32     preOffset = cur;
33   }
34 }
```

Figure 38. GA implementation in Pāñini (continued). These handlers all conflict with themselves.

dlers are $\{\textbf{read}\ \mathit{GenCont\ \ done}, \textbf{ann}\ \mathit{GenReady}\}$. For `CrossOver`, the read effect comes from the field read on line 12 and the announce effect comes from the **announce** expression on line 17. On line 15, applying the crossover function on a new generation has no effect. That is because `g1` was created on line 13 and all the changes to this local copy may not be visible to other methods until it escapes [58] out of the method on line 13.

A static type-and-effect system would yield too conservative of results for this application. The effect set would at least include the effects of the handlers shown in Figure 38 and these effects conflict with each other: these handlers methods have an instance field write effect and any one of them does not commute with itself [47]. Therefore, it is unsafe to concurrently execute `crossover` and `mutation`. However, not all usage of this genetic algorithm framework will require all of these handlers and most of the usage do not. Also, almost all the fitness functions in JGAP [39], except some special cases like the one in Figure 38, are pure functions. Thus, our hybrid type-and-effect system is likely to be very useful towards exposing concurrency in this genetic algorithm implementation.

6.6.1 Experimental Setup and Results

We implemented this GA and ran two versions of it, one that registers a conflicting logger and another that does not, to observe the speedup from concurrently executing the handlers `Mutation` and `CrossOver`. For this experiment, the generation (or population) size was 3000 and the depth (number of generations) was 10. Figure 39 shows the results.

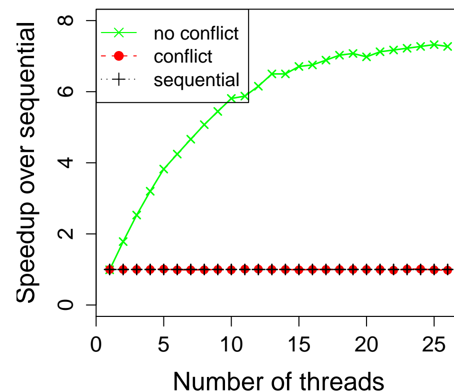

Figure 39. GA: speedup over sequential version. Speedup scales to 7.3x as number of threads increases.

6.6.2 Analysis of results

As expected, the version with no conflict shows good speedup (considering the concurrency available), achieving 7.3x speedup, while the version with conflicts was serialized. On the other hand, with a static type-and-effect system both of these scenarios would have been serialized, because

the schedule produced at compile time must be serial if the handlers conflict for any input.

7. Evaluation: Overhead of Effect System

This section evaluates our second hypothesis. That is that our hybrid type-and-effect system has acceptable overhead. We now consider the dynamic overhead of the type-and-effect system for real world applications. Recall that this overhead occurs upon handler registration.

7.1 FindBugs

First, we measured the overhead of registration for the FindBugs system. To do so, we compared the runtime for registrations against the overall application runtime. The experimental setup is similar to that described in Section 6.2. However, in this experiment we ran FindBugs with three different input workloads: 1 application, 10 applications and 50 applications. These results are shown in Figure 40.

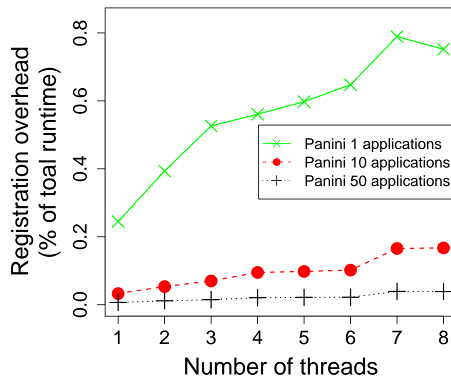

Figure 40. Registration overhead for FindBugs. Handlers have no conflicts. Less than 0.04% overhead for large input.

The results show that registration makes up only a small portion of the overall execution time (less than 0.8%) even for small workloads. The overhead increases slightly as the number of threads increases because the execution time of the entire program decreases. However, for larger workloads, even with many threads, overhead is less than 0.04%.

7.2 jASEN: Anti Spam ENgine

Next, we measured the overhead of registration for jASEN, a spam detection system. To do so, we compared the runtime of registrations to that of the rest of the spam detection. We ran the spam detection system with different workload sizes: 1, 10, and 30 messages. These results are shown in Figure 41.

When 1 message is analyzed, the overhead is significant (around 14%). However, the overhead decreased significantly when 10 messages were processed and dropped to less than 2% when 30 messages were analyzed. Considering that a typical e-mail system will handle far more than 30 messages, this overhead is acceptable. Also, the gains

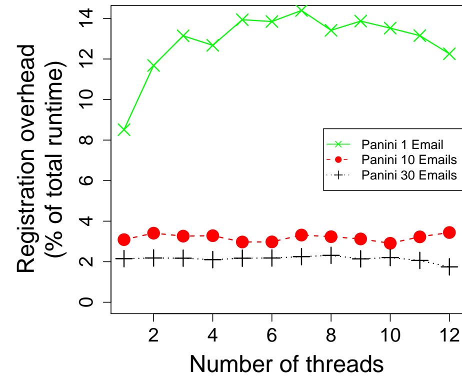

Figure 41. Registration overhead for jASEN. Handlers have no conflicts. Less than 2% overhead for large input.

achieved by parallelizing this system (as shown in Figure 23) outweigh the costs of registration.

7.3 Refactoring Crawler

Next, we measured the overhead of registration for the refactoring crawler application. We ran the application with three different input workloads: applications JHotDraw, Struct, and Eclipse. These results are shown in Figure 42.

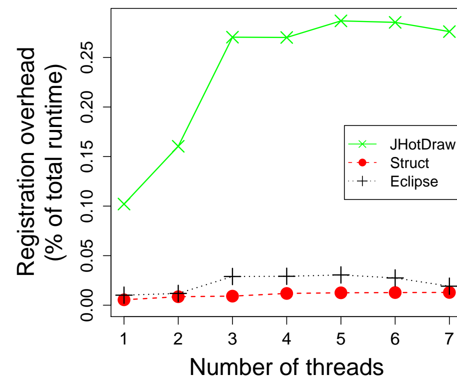

Figure 42. Registration overhead for Refactoring Crawler. Handlers have no conflicts. Only approximately 0.01% overhead for large workload.

Again, we see expected results in that registration takes only a very small portion of the execution time. In this case we see only approximately 0.3% overhead even for many threads for the smallest workload. For the larger workload sizes, we see only about 0.01% overhead for many threads.

7.4 Web Crawler

Next, we measured the overhead for the WebSPHINX web crawler program. We tested with four different crawling depths. These results are shown in Figure 43.

Notice that even for small web-crawling depths, the overhead is only about 0.25%. For larger depths the overhead becomes negligible (about 0.01%).

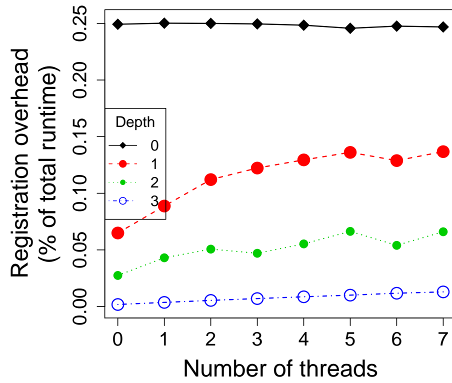

Figure 43. Registration overhead for WebSPHINX. Less than 0.01% overhead for modest crawling depths.

7.5 Genetic Algorithm

Finally, we measured the overhead of registration for the genetic algorithm application. In this experiment, we use a similar setup to the previous experiment except that we also measured overhead for depths eight and nine (on top of the original depth of 10). These results are shown in Figure 44.

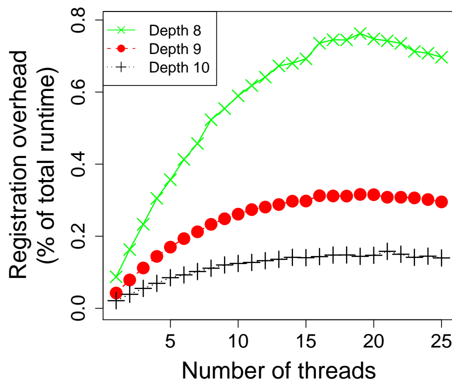

Figure 44. Registration overhead for GA. Handlers have no conflicts. Roughly 0.1% overhead for large generation depth.

As expected, the registration takes only a very small portion of the execution time. We see less than 0.8% overhead for a depth of eight. Like earlier experiments, this overhead increases as the number of threads increases because the execution time of the entire program decreases. For the largest depth (10) we saw only about 0.1% overhead.

Summary. We have shown, through the study of real world applications, the potential utility of our technique towards exposing implicit concurrency in shared-memory II programs. These results show that for real world applications our claims hold. These claims are that

- our hybrid analysis is more precise compared to a fully static analysis resulting in greater concurrency, and
- our hybrid analysis has negligible overhead that is amortized by the introduced concurrency.

8. Related Work

Pāñini. This work builds on our previous work on the Pāñini language [36] in several ways.

First, we give a calculus describing the hybrid effect analysis model and formalize the semantics of Pāñini. This has enabled a rigorous understanding of key concurrency properties (e.g. deadlock and data race freedom, etc.). These aspects of Pāñini differ from previous approaches. This calculus has also enabled verification of the implementation and testing of semantic variations. This includes the ability to plug in more powerful static analysis techniques. Such improvements would be important in practical settings.

Next, we provide a thorough discussion and experimental evaluation of Pāñini's hybrid effect analysis. This includes a study using real world programs of our hybrid type-and-effect system as compared to static approaches. Our evaluation has answered several questions about the runtime performance of the hybrid analysis in various situations.

In summary, this work makes important contributions over previous work both theoretically and practically. The theoretical advances have enabled us to verify key concurrency properties, present a more clearly defined language design, and consider practical future improvements for Pāñini. The practical advances have answered key questions regarding the benefits of Pāñini's hybrid type-and-effect system and the overall practical use of Pāñini.

Types, Regions and Effects. Pāñini's hybrid system is not the first to use type-and-effect to enable safe concurrency. Deterministic parallel Java (DPJ/DPJizer) [45, 55] uses a region-based type-and-effect system to provide deterministic parallelism in imperative OO programs. Ownership systems [13–15] have been used to organize objects into hierarchies for better reasoning, i.e. about the absence of aliasing. Concurrent Revisions [12] provides users with a syntax that says each thread accesses its own version of certain objects to eliminate interferences. Similar to these analyses, our hybrid system generates static invariants, e.g. effect summaries for every method. To the best of our knowledge, compared to these related ideas, Pāñini's type-and-effect system is the first that effects in a hybrid manner. Next, the schedule produced by purely static approaches must be valid for all inputs and thus may declare many programs concurrency-unsafe, even though only certain rare control flow paths in such programs produce concurrency-unsafe computational effects. Our hybrid system computes schedules during program execution where it has more accurate information. Therefore, it may observe more safe parallelization opportunities than the purely static approaches.

Atomicity. Several systems provide syntax to declare and validate the atomicity of certain data structures and thus guarantee concurrency safety. AJ2 and Rcc/Sat [23, 24] use the type system to enforce certain locking disciplines to check for data races or ensure atomic access objects. AJ [56]

uses a type system to maintain data-centric synchronization and is a variant of the atomicity protection. Unlike these works, our hybrid system ensures a deterministic semantics, not just atomicity. Sometimes, atomicity is not enough to guarantee a deterministic semantics. For example, although it is safe to concurrently add elements to a list, the order of the insertions is violated and could be arbitrary. Second, our focus is on producing and validating a safe schedule, while they offer constructs for programmers to facilitate the reasoning on concurrency safety, statically. Therefore, these works are orthogonal to ours. Thus our hybrid system may enhance its accuracy by combining with these static techniques.

Dynamic Approaches. Dynamic approaches are also used to ensure concurrency safety. The Galois system [32, 40] aims to optimistically parallelize irregular applications. Central to this system is a worklist where pending operations live and more operations can be inserted into this list. Their underlying implementation uses speculative execution. In contrast to this, our system infers the computational effects for operations statically and then uses these effects at runtime to determine a safe schedule for operation execution that maximizes concurrency. The Galois system requires use of a thread speculation infrastructure at runtime, e.g. to implement a rollback mechanism, whereas our effect system requires an effect manipulation and a scheduling mechanism. Furthermore, unlike previous work on dynamic approaches [44, 57], our work does not require modifications to the underlying virtual machine.

Actor-based Languages. There is a large body of work on using the notion of actors [5] for concurrency. Agha and Hewitt's work [6] and Erlang's language design [7] model programs as a set of "isolated" active entities that communicate by passing messages. JCoBox [51] unifies the actor model with the shared memory model to enhance local and distributed concurrency. In these models, actors process local computations concurrently with other actors. The actor model is seen as naturally supporting concurrency. Also, complete isolation of actors makes it easier to reason about their states. However, in mainstream object-oriented languages such as Java, C++, etc., programmers rely on shared states to express many useful computational idioms. So although in principle it would be sensible to adopt a fully-isolated actor-based model, practice and existing investment in mainstream languages demands a solution that supports both message-passing and shared states. Also, our system guarantees a deterministic semantics, which is somewhat difficult for the actor model, due to asynchronous and non-deterministic nature of the message passing paradigm [38].

Event-based Systems. Events have a long history in both the software design [22, 34] and distributed systems communities [20]. Pāñini's notion of asynchronous, typed events builds on these notions, in particular recent work in programming languages focusing on event-driven design [18, 19, 46]. Pāñini's design is not the first to integrate event-based model with concurrency. Reactor [52] pattern integrates the demultiplexing of events and the dispatching of the corresponding event handlers to simplify event-driven applications. Li and Zdancewic [34] promote the integration of event-based model with the thread-based explicit concurrency models. TaskJava [22] provides syntax to mark `asynchronous` methods. Expressions may express their interests in a set of events and the expressions will block until one of them fires. Pāñini's design is also not the first to promote implicit concurrency, e.g. in BETA [53], objects implicitly execute in the context of a local process.

The above models, developed for event-based distributed systems, assume that components in the system do not share state and only communicate by passing primitive values, whereas Pāñini allows shared states (similar to mainstream languages like Java, C#), which is useful for many computation patterns. Also, unlike the above works on shared memory [8, 22, 34], Pāñini provides safety guarantees. Pāñini provides programmers with deterministic semantics via its hybrid type-and-effect system. As a result, programmers are relieved of reasoning about concurrency bugs. Such software engineering properties are becoming very important with the increasing presence of concurrent software, increasing interleaving of threads in concurrent software, and increasing number of under-prepared software developers writing code using concurrency unsafe features.

9. Conclusion and Future Work

The implicit invocation (II) design style is widely used in mainstream shared-memory languages, e.g. via the observer design pattern [26]. Thus, language features that promote safe concurrency for the II design style have become important [18, 42, 52]. Static type-and-effect systems [14, 45] are effective at eliminating data races and deadlocks in explicitly concurrent languages, however, they are often too conservative and reject programs written in II style where concurrency could be safely exposed. The actor model [5] exposes concurrency by providing a disjoint memory model. However, due to the asynchronous nature of the message passing model, it does not provide a deterministic semantics. Also, programmers that are well-versed in mainstream OO languages have to make great efforts to adapt to this model.

We have developed a new hybrid type-and-effect system that solves these problems. This system is based on our observations that handler registrations are infrequent compared to event announcements and that the exact set of tasks to be run when an event is signaled can be computed during registration. We have shown several real world applications where our type-and-effect system exposes more concurrency than completely static type-and-effect systems. The results gathered from running these applications have shown that the overhead of our effect system is acceptable and its per-

formance benefits are promising. Finally, our effect system provides race and deadlock freedom and a deterministic semantics.

In this work, we have deliberately avoided aliasing issues by keeping the read and write effects limited. This allowed us to focus on the announcement and registration effects. In the future, we would like to explore the integration of our type-and-effect system with an ownership type system [13–15] to further enhance its precision and effectiveness.

References

[1] FindBugs Eclipse Plugin. `http://marketplace.eclipse.org/content/findbugs-eclipse-plugin`.

[2] The Java Anti Spam ENgine. `http://www.jasen.org/`.

[3] Java Line Count Estimator. `http://reasoning.com/downloads/java_line_count_estimator.html`.

[4] M. Abadi and G. Plotkin. A model of cooperative threads. In *POPL*, pages 29–40, 2009.

[5] G. Agha. Actors: a model of concurrent computation in distributed systems. Technical Report AITR-844, MIT, 1985.

[6] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41. Springer, 1985.

[7] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.

[8] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5): 769–804, 2004.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA*, pages 81–96, 2009.

[10] R. Bocchino. *An Effect System and Language for Deterministic-by-Default Parallel Programming*. PhD thesis, University of Illinois, Urbana-Champaign, 2010.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[12] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, pages 691–707, 2010.

[13] N. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA*, pages 618–633, 2010.

[14] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *OOPSLA*, pages 441–460, 2007.

[15] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[16] C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *Science of Computer Programming*, 63 (3):321–374, 2006.

[17] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated Detection of Refactorings in Evolving Components. In *ECOOP*, pages 404–428, 2006.

[18] P. Eugster. Type-Based Publish/Subscribe: Concepts and Experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.

[19] P. Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *ECOOP*, pages 570–584, 2009.

[20] P. T. Eugster, R. Guerraoui, and C. H. Damm. On Objects and Events. In *OOPSLA*, pages 254–269, 2001.

[21] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *SPAA*, pages 1–11, 1997.

[22] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *PEPM*, pages 134–143, 2007.

[23] C. Flanagan and S. N. Freund. Type inference against races. *Sci. Comput. Program.*, 64:140–165, 2007.

[24] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI*, pages 47–58, 2005.

[25] M. Flatt, S. Krishnamurthi, and M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. Springer, 1999.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[27] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91*, pages 31–44, 1991. ISBN 3-540-54834-3.

[28] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38, 1986.

[29] B. Goetz, T. Peierls, B. J., J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[30] R. T. Hammel and D. K. Gifford. Fx-87 performance measurements: Dataflow implementation. Technical report, Cambridge, MA, USA, 1988.

[31] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, December 2004.

[32] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[33] D. Lea. A Java Fork/Join Framework. In *Java Grande*, pages 36–43, 2000.

[34] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI*, pages 189–199, 2007.

[35] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP*, pages 158–167, 2006.

[36] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE*, 2010.

[37] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

[38] P. Mackay. Why has the actor model not succeeded? Technical Report 2, Imperial College, Dept. of Comput., 1997.

[39] K. Meffert. JGAP - Java Genetic Algorithms and Genetic Programming Package. `http://jgap.sf.net`.

[40] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443, 2010.

[41] R. C. Miller and K. Bharat. Sphinx: a framework for creating personal, site-specific web crawlers. In *WWW*, pages 119–130, 1998.

[42] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. In *SOSP*, pages 58–68, 1993.

[43] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP*, pages 1–12, 2003.

[44] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *OOPSLA*, pages 206–223, 2004.

[45] R. Bocchino *et al.*. A type and effect system for deterministic parallel java. In *OOPSLA*, pages 97–116, 2009.

[46] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[47] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis framework for parallelizing compilers. In *PLDI*, pages 54–67, 1996.

[48] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[49] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, pages 394–409, 2009.

[50] R. D. Salcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *In VMCAI*. Springer-Verlag, 2005.

[51] J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *ECOOP*, pages 275–299. Springer, June 2010.

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

[53] B. Shriver and P. Wegner. Research directions in object-oriented programming, 1987.

[54] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111:245–296, 1994.

[55] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring method effect summaries for nested heap regions. In *ASE*, pages 421–432, 2009.

[56] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.

[57] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2005.

[58] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, pages 187–206. ACM, 1999.

[59] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994.

Appendix: Omitted Semantics Details

The auxiliary function *updateHierarchy* (defined in Figure 45) is responsible for updating the hierarchy upon a registration. We now briefly define each step in this process.

$$
\begin{array}{ll}
\mathit{updateHierarchy}(loc,\gamma,\mu) = \gamma & \textbf{if } \mathit{registered}(loc,\gamma) = \mathit{true} \\
\mathit{updateHierarchy}(loc,\gamma,\mu) = \gamma' & \textbf{if } \mathit{registered}(loc,\gamma) = \mathit{false} \\
\quad \textbf{where } \mu(loc) = [c.F], \quad \mathit{colEvents}(c, CT) = P, & \\
\quad \gamma'' = \mathit{putEvent}(P, loc, \gamma, \mu), & \\
\quad \gamma''' = \mathit{updateEvMap}(\gamma''), \quad \gamma' = \mathit{reorderEvMap}(\gamma''') &
\end{array}
$$

$$
\begin{array}{l}
\mathit{registered}(loc,\gamma) = t \\
\quad \textbf{where } \gamma = \{p_i \mapsto \delta_i\}, \quad t = \bigcup_i \mathit{inHierarchy}(loc, \delta_i)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{inHierarchy}(loc, \bullet) = \mathit{false} & \\
\mathit{inHierarchy}(loc, \zeta + \delta) = \mathit{inHierarchy}(loc, \delta) & \textbf{if } \mathit{inList}(loc, \zeta) = \mathit{false} \\
\mathit{inHierarchy}(loc, \zeta + \delta) = \mathit{true} & \textbf{if } \mathit{inList}(loc, \zeta) = \mathit{true}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{inList}(loc, \bullet) = \mathit{false} & \\
\mathit{inList}(loc, \langle loc', \rho\rangle + \zeta) = \mathit{inList}(loc, \zeta) & \textbf{if } loc \neq loc' \\
\mathit{inList}(loc, \langle loc, \rho\rangle + \zeta) = \mathit{true} & \textbf{otherwise}
\end{array}
$$

Figure 45. Auxiliary functions for registering a handler.

1. First, the function checks whether the handler has registered before (using *registered*). If it has registered before, the configuration does not change. For simplicity, Pāñini does not allow multiple registrations for the same object.
2. If this handler has not registered before, it searches the class table to collect all of the events that this handler subscribes to. This is done using the *colEvents* function (defined in Figure 46).
3. Next, the handler is put into the hierarchy for each of these events (those found by *colEvents*) using the *putEvent* function defined in Figure 47.
4. The *updateEvMap* function (defined in Figure 48) is then used to update the effects for each handler based on the new registration. This is performed until a fix-point is reached. The purpose of this step is to propagate the effects of this new handler to all handlers which may cause this handler to run.
5. Finally, the *reorderLvl* function is used to reorder the hierarchy to guarantee that no handlers which may conflict may be run in parallel.

The function *inHierarchy* checks if a handler is already in the event hierarchy and the function *inList* checks if a handler is in a specific level of the event hierarchy.

$$
\begin{array}{ll}
\mathit{colEvents}(c, \bullet) = \bullet & \\
\mathit{colEvents}(c, (\texttt{event}\ p\{\ldots\}) + CT') = \mathit{colEvents}(c, CT') & \\
\mathit{colEvents}(c, (\texttt{class}\ c' \ldots) + CT') = \mathit{colEvents}(c, CT') & \textbf{if } c \neq c' \\
\mathit{colEvents}(c, ((\texttt{class}\ c\ \texttt{extends}\ d\ \ldots\ \mathit{binding}_1 \ldots \mathit{binding}_n) + CT')) & \\
= \{\mathit{colEvent}(\mathit{binding}_n)\} \cup \ldots \cup \{\mathit{colEvent}(\mathit{binding}_1)\} \cup \mathit{colEvents}(d, CT) & \\
\quad \textbf{where } \mathit{colEvent}(\texttt{when}\ p\ \texttt{do}\ m) = p &
\end{array}
$$

Figure 46. Functions for collecting events of interest.

Next, consider the *colEvents* function which is defined in Figure 46. Recall that this function is responsible for searching class table to collect all the events a handler is interested in. It will first search the binding declarations in the handlers' class and recursively search its super classes.

$$
\begin{array}{l}
\mathit{putEvent}(\emptyset, loc, \gamma, \mu) = \gamma \\
\mathit{putEvent}(p + P, loc, \gamma, \mu) = \gamma'' \\
\quad \textbf{where } \delta = \gamma(p), \quad \mathit{putHierarchies}(loc, \delta, \mu, p) = \delta', \\
\quad \gamma' = \gamma \uplus \{p \mapsto \delta'\}, \quad \gamma'' = \mathit{putEvent}(P, loc, \gamma', \mu)
\end{array}
$$

$$
\begin{array}{l}
\mathit{putHierarchies}(loc, \delta, \mu, p) = \zeta + \delta \\
\quad \textbf{where } \mu(loc) = [c.F], \quad \mathit{hmatch}(c, p, CT) = m, \\
\quad (c', t, m \ldots, \rho) = \mathit{findMeth}(c, m), \\
\quad \zeta = \langle loc, m, \rho\rangle + \bullet
\end{array}
$$

Figure 47. Adding a handler into an event hierarchy.

Now, consider the functions defined in Figure 47. The *putEvent* function uses the *putHierarchies* function to put current handler ($p \in P$) into the hierarchies of all the events that this handler is interested in.

$$
\begin{array}{ll}
\mathit{updateEvMap}(\gamma) = \gamma & \textbf{if } \mathit{fixpEvMap}(\gamma) = \langle \gamma', \mathit{true}\rangle \\
\mathit{updateEvMap}(\gamma) = \mathit{updateEvMap}(\gamma') & \textbf{if } \mathit{fixpEvMap}(\gamma) = \langle \gamma', \mathit{false}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{fixpEvMap}(\gamma) = \langle \gamma', t\rangle \\
\quad \textbf{where } \gamma' = \{p_i \mapsto \delta'_i \mid p_i \in \mathit{dom}(\gamma) \wedge \gamma(p_i) = \delta_i \wedge \\
\qquad\qquad \langle \delta'_i, t_i\rangle = \mathit{fixpHier}(\delta_i, \gamma)\}, \\
\quad T = \{t_i \mid \exists p_i \in \mathit{dom}(\gamma) \text{ s.t. } \gamma(p_i) = \delta_i \wedge \langle \delta'_i, t_i\rangle = \mathit{fixpHier}(\delta_i, \gamma)\}, \\
\quad t = \bigwedge_{t_i \in T} t_i
\end{array}
$$

$$
\begin{array}{l}
\mathit{fixpHier}(\bullet, \gamma) = \langle \bullet, \mathit{true}\rangle \\
\mathit{fixpHier}(\zeta + \delta, \gamma) = \langle \emptyset, t''\rangle \\
\quad \textbf{where } \mathit{fixpEpsilon}(\zeta) = \langle \zeta', t\rangle, \\
\quad \mathit{fixpHier}(\delta, \gamma) = \langle \delta', t'\rangle, \quad t'' = t' \wedge t
\end{array}
$$

$$
\begin{array}{l}
\mathit{fixpLevel}(\bullet, \gamma) = \langle \bullet, \mathit{true}\rangle \\
\mathit{fixpLevel}(\iota + \zeta, \gamma) = \langle \iota' + \zeta', t''\rangle \\
\quad \textbf{where } \mathit{fixpEpsilon}(\iota) = \langle \iota', t\rangle, \\
\quad \mathit{fixpLevel}(\zeta, \gamma) = \langle \zeta', t'\rangle, \quad t'' = t' \wedge t
\end{array}
$$

$$
\begin{array}{l}
\mathit{fixpEpsilon}(\langle loc, m, \rho\rangle, \gamma) = \langle \langle loc, m, \rho'\rangle, \rho = \rho'\rangle \\
\quad \textbf{where } P = \{p_i \mid \exists\, \epsilon_j \in \rho \text{ s.t. } \epsilon_j = \textbf{ann}\ p_i\}, \\
\quad \rho' = \rho \bigcup_{p_i \in P} \mathit{effHierachy}(\gamma(p_i))
\end{array}
$$

Figure 48. Functions for updating effects for handlers.

Next, consider the fix-point update functions defined in Figure 48. The top level *updateEvMap* function is used to update the effects of each handler using a fix-point algorithm. This function will repeatedly call the *fixpEvMap* function until the hierarchies no longer change. For each call to *fixpEvMap* each event hierarchy (δ) is updated using the *fixpHier* function which uses the *fixpLevel* function to update each level of the hierarchy. This function uses the function *fixpEpsilon* to update each announcement effect. For each event p, that it may announce, it merges the effects of all the handlers of event p into the effect of the current task. This algorithm is guaranteed to terminate because the total effect could only be the union of all the effects of all the handlers. And in each iteration, there is at least one handler whose effect is changed.

Finding the effects of all the handlers of an event p is done using the function *effHierachy* (defined in Figure 49) which merges the effects of all handlers in each level. The effect of a handler is found using the *effList* function.

$$
\begin{aligned}
&\mathit{effHierachy}(\bullet) = \emptyset\\
&\mathit{effHierachy}(\zeta + \delta) = \mathit{effList}(\zeta) \cup \mathit{effHierachy}(\delta)\\[1em]
&\mathit{effList}(\bullet) = \emptyset\\
&\mathit{effList}(\langle loc, m, \rho\rangle + \zeta) = \rho \cup \mathit{effList}(\zeta)
\end{aligned}
$$

Figure 49. Computing effects for the entire hierarchy.

Since the effects in each of the event hierarchies can be updated, the handlers in the hierarchy need to be reordered to eliminate potential data races. This is accomplished using the $\mathit{reorderEvMap}$ function which updates all event hierarchies. The $\mathit{reorderHier}$ function takes a specific event hierarchy and re-orders each level using the $\mathit{reorderLvl}$ function. The $\mathit{reorderLvl}$ function takes the first handler in the original list, compares its updated effect set with all the handlers in the last level in the updated list (by the comp function). If there is no conflict, it is put into this level. Otherwise, it is put into a new level. The boolean variable t in these functions is used to determine whether there is a handler h that registered before that has a **`reg`** effect. If such a handler exists, we make the later registered handler, h', which has an **`ann`** , conflict with the other handlers. This is because during the execution of h, more handlers could register. The **`ann`** effect for h' maybe be enlarged and h' may conflict with handler h'' even though h' and h'' did not conflict originally.

$$
\begin{aligned}
&\mathit{reorderEvMap}(\gamma) = \gamma'\\
&\quad \textbf{where } \gamma' = \{p_i \mapsto \delta'_i \mid p_i \in \mathit{dom}(\gamma) \wedge \gamma(p_i) = \delta_i \wedge\\
&\qquad\qquad\qquad \delta'_i = \mathit{reorderHier}(\delta_i, \bullet, \mathit{false})\\[1em]
&\mathit{reorderHier}(\bullet, \delta, t) = \delta\\
&\mathit{reorderHier}(\delta' + \zeta, \delta, t) = \mathit{reorderHier}(\delta', \delta'', t')\\
&\quad \textbf{where } \mathit{reorderLvl}(\zeta, \delta, t) = \langle \delta'', t'\rangle\\[1em]
&\mathit{reorderLvl}(\bullet, \delta, t) = \langle \delta, t\rangle\\
&\mathit{reorderLvl}(\zeta + \iota, \bullet, t) = \mathit{reorderLvl}(\zeta, \zeta', t)\\
&\quad \textbf{where } \zeta' = \iota + \bullet, \qquad \iota = \langle loc, m, \rho\rangle, \qquad t = \mathit{hasReg}(\rho)\\
&\mathit{reorderLvl}(\zeta + \iota, \zeta' + \delta, t) = \mathit{reorderLvl}(\zeta, \delta', t) \quad \textbf{if } \mathit{comp}(\zeta', \iota, t) = \mathit{true}\\
&\quad \textbf{where } \zeta'' = \iota + \zeta',\ \delta' = \zeta'' + \delta,\ \iota = \langle loc, m, \rho\rangle,\ t' = t \mathbin{||} \mathit{hasReg}(\rho)\\
&\mathit{reorderLvl}(\zeta + \iota, \zeta' + \delta, t) = \mathit{reorderLvl}(\zeta, \delta', t) \quad \textbf{if } \mathit{comp}(\zeta', \iota, t) = \mathit{false}\\
&\quad \textbf{where } \zeta'' = \iota + \bullet,\ \delta' = \zeta'' + \zeta' + \delta,\ \iota = \langle loc, m, \rho\rangle,\ t' = t \mathbin{||} \mathit{hasReg}(\rho)\\[1em]
&\mathit{comp}(\bullet, \iota, t) = \mathit{true}\\
&\mathit{comp}(\langle loc', m', \rho'\rangle + \zeta, \langle loc, m, \rho\rangle, t) = \mathit{indep}(\rho, \rho', t) \wedge\\
&\qquad\qquad\qquad \mathit{comp}(\zeta, \langle loc, m, \rho\rangle, t)\\[1em]
&\mathit{hasReg}(\bullet) = \mathit{false}\\
&\mathit{hasReg}(\epsilon + \rho) = \mathit{hasReg}(\rho) \qquad \textbf{if } \epsilon \neq \mathtt{\textbf{reg}}\\
&\mathit{hasReg}(\epsilon + \rho) = \mathit{true} \qquad\quad\;\; \textbf{if } \epsilon = \mathtt{\textbf{reg}}
\end{aligned}
$$

Figure 50. Functions for reordering the event hierarchies.

Figure 51 defines the functions responsible for building event hierarchies. The spawn function is used when an event is announced. It first pulls the event hierarchy from the event map γ. Each level is build using the $\mathit{buildLevel}$ function which creates tasks using the $\mathit{buildTask}$ function. In the end, each new task depends on all the tasks in the previous level of the hierarchy.

$$
\begin{aligned}
&\mathit{spawn}(p, \psi, \gamma, \nu, \mu) = \langle \gamma' + \gamma, I\rangle\\
&\quad \textbf{where } \delta = \gamma(p), \qquad \mathit{buildHier}(\delta, \mu, \emptyset, \nu) = \langle \gamma', I\rangle\\[1em]
&\mathit{buildHier}(\bullet, \mu, I, \nu) = \langle \bullet, \emptyset\rangle\\
&\mathit{buildHier}(\delta + \zeta, \mu, I, \nu) = \langle \psi + \psi', I' \cup I''\rangle\\
&\quad \textbf{where } \mathit{buildLevel}(\zeta, \mu, I, \nu) = \langle \psi, I'\rangle,\\
&\quad \mathit{buildHier}(\zeta, \mu, I', \nu) = \langle \psi', I''\rangle\\[1em]
&\mathit{buildLevel}(\bullet, \mu, I, \nu) = \langle \bullet, \emptyset\rangle\\
&\mathit{buildLevel}(\iota + \zeta, \mu, I, \nu) = \langle \psi', I'''\rangle\\
&\quad \textbf{where } \mathit{buildTask}(\iota, \mu, I, \nu) = \langle e, \langle id, I\rangle\rangle,\\
&\quad \mathit{buildLevel}(\zeta, \mu, I, \nu) = \langle \psi, I'\rangle,\\
&\quad \psi' = \langle e, \langle id, I\rangle\rangle + \psi, \qquad I'' = I' \cup \{id\}\\[1em]
&\mathit{buildTask}(\langle loc, m, \rho\rangle, \mu, I, \nu) = \langle e', \langle id, I\rangle\rangle\\
&\quad \textbf{where } \mu(loc) = [c.F], \qquad e' = [\mathtt{\textbf{this}}/loc, var_1/v_1, \ldots, var_n/v_n]e,\\
&\quad (c', t, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}, \ldots) = \mathit{findMeth}(c, m),\\
&\quad \nu = (v_1, \ldots, v_n), \qquad id = \mathit{fresh}()
\end{aligned}
$$

Figure 51. Functions for creating task configurations.