

# Pāṇini: Reconciling Concurrency and Modularity in Design

Yuheng Long, Sean L. Mooney, Tyler Sondag and Hridesh Rajan

Initial Submission: March 25, 2010.

**Keywords:** 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) 2010, Yuheng Long, Sean L. Mooney, Tyler Sondag and Hridesh Rajan.

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

# Pāñini: Reconciling Concurrency and Modularity in Design

Yuheng Long   Sean L. Mooney   Tyler Sondag   Hridesh Rajan

Dept. of Computer Science, Iowa State University  
{csgzlong,smooney,sondag,hridesh@iastate.edu}

## Abstract

Writing correct and efficient concurrent programs still remains a challenge. Explicit concurrency is difficult, error prone, and creates code which is hard to maintain and debug. This type of concurrency also treats modular program design and concurrency as separate goals, where modularity often suffers. To solve these problems, we are designing a new language that we call Pāñini. In this paper, we focus on Pāñini’s *asynchronous, typed events* which reconcile the modularity goal promoted by the implicit invocation design style with the concurrency goal of exposing potential concurrency between the execution of subjects and observers. Since modularity is improved and concurrency is implicit in Pāñini, programs are easy to reason about and maintain. Furthermore, races and deadlocks are avoided entirely yielding programs with a guaranteed sequential semantics. To evaluate our language design and implementation we show several examples of its usage as well as an empirical study of program performance. We found that not only is developing and understanding Pāñini programs significantly easier compared to standard concurrent object-oriented programs, but performance of Pāñini programs is comparable to the equivalent programs written using Java’s fork-join framework.

## 1. Introduction

*Coming together is a beginning. – Henry Ford*

The idea behind Pāñini’s design is that if programmers structure their system to improve modularity in its design, they should get concurrency for free.

### 1.1 Object-oriented (OO) Concurrency Features

It is widely accepted that multicore computing is becoming the norm. However, writing correct and efficient concurrent programs using *concurrency-unsafe* features remains a challenge [5, 33–35, 47]. A language feature is concurrency-unsafe if its usage may give rise to program execution sequences containing two or more memory accesses to the same location that are not ordered by a happens-before relation [26]. Several such language features exist in common language libraries, for example, threads, processes, Futures, and FutureTask are all examples from the

Java programming language’s standard library [35, 47]<sup>1</sup>. Using such language features has advantages, e.g. they can encapsulate complex synchronization code and allow its reuse.

To illustrate, consider the implementation of a genetic algorithm in Java presented in Figure 1. The idea behind a genetic algorithm is to mimic the process of natural selection. Genetic algorithms are computationally intensive and are useful for many optimization problems [42]. The main concept is that searching for a desirable state is done by combining two *parent* states instead of modifying a single state [42]. An initial *generation* with  $n$  members is given to the algorithm. Next, a *cross over* function is used to combine different members of the generation in order to develop the next generation (lines 10–16 in Figure 1). Optionally, members of the offspring may randomly be *mutated* slightly (lines 18–23 in Figure 1). Finally, members of the generation (or an entire generation) are ranked using a *fitness function* (lines 25–29 in Figure 1).

**Multiple Concerns of the Genetic Algorithm.** In the OO implementation of the genetic algorithm in Figure 1 there are three concerns standard to the genetic algorithm: cross over (creating a new generation), mutation (random changes to children), and fitness calculation (how good is the new generation). Logging of each generation is another concern added here since it may be desirable to observe the space searched by the algorithm (lines 17 and 24). The final concern here is concurrency (lines 4, 7–9, and 30–33). In this example, production of a generation is run as a FutureTask. The different shading shows code corresponding to each concern as illustrated in the legend.

### 1.2 Problems with Explicit Concurrency Features

**Explicit concurrency.** With explicit concurrency, the programmer must divide the program into independent tasks. Next, the programmer must handle creating and managing the individual threads. A problem with the concurrency-unsafe language features described previously and illustrated in Figure 1 is that correctness is difficult to ensure since it relies on all objects obeying a usage policy [27]. Since such policies cannot automatically be enforced by a library based approach [27], the burden on the programmer is in-

<sup>1</sup>Original proposal for futures in MultiLisp is concurrency-safe [41].

Legend	Concurrency	Logging	Mutation	Cross over	Fitness
--------	-------------	---------	----------	------------	---------

```

1 class GeneticAlgorithm {
2   float crossOverProbability, mutationProbability;
3   int max;
4   ExecutorService executor;
5   //Constructor elided (Initializes fields above).
6   public Generation compute(final Generation g) {
7     FutureTask<Generation> t = new FutureTask<Generation>(
8       new Callable<Generation>() {
9         Generation call() {
10          int genSize = g.size();
11          Generation g1 = new Generation(g);
12          for (int i = 0; i < genSize; i += 2) {
13            Parents p = g.pickParents();
14            g1.add(p.tryCrossOver(crossOverProbability));
15          }
16          if(g1.getDepth() < max) g1 = compute(g1);
17          logGeneration(g1);
18          Generation g2 = new Generation(g);
19          for (int i = 0; i < genSize; i += 2) {
20            Parents p = g.pickParents();
21            g2.add(p.tryMutation(mutationProbability));
22          }
23          if(g2.getDepth() < max) g2 = compute(g2);
24          logGeneration(g2);
25          Fitness f1 = g1.getFitness();
26          Fitness f2 = g2.getFitness();
27          if(f1.average() > f2.average()) return g1;
28          else return g2;
29        }
30      });
31      executor.execute(t);
32      try { return t.get(); }
33      catch (InterruptedException e) { return g; }
34      catch (ExecutionException e) { return g; }
35    }
36  }

```

Figure 1. Genetic algorithm with Java concurrency utilities

created and errors arise (ex: deadlock, data races, etc.). Also, the non-determinism introduced by such mechanisms makes debugging hard since errors are difficult to reproduce [46]. Furthermore, this style of explicit parallelism can hurt the design and maintainability of the resulting code [40].

**Separation of modular and concurrent design.** Another significant shortcoming of these language features, or perhaps the discipline that they promote, is that they treat modular program design and concurrent program design as two separate and orthogonal goals.

From a quick glance at Figure 1, it is quite clear that the five concerns are tangled. For example, the code for concurrency (lines 4, 7-9, and 30-33) is interleaved with the logic of the algorithm (the other four concerns). Also, the code for logging occurs in two separate places (lines 17 and 24). This arises from implementing a standard well understood sequential approach and then afterward attempting to expose concurrency rather than pursuing modularity and concurrency simultaneously. Aside from this code having poor modularity, it is not immediately clear if there is any potential concurrency between the individual concerns (cross over, mutation, logging, and fitness calculation).

```

1 event GenAvailable {
2   Generation g;
3 }
4 class CrossOver {
5   Number probability; Number max;
6   init(...){
7     register(this)
8     // initialization elided (initializes fields above).
9   }
10  when GenAvailable do cross;
11  void cross(Generation g) {
12    Number gSize = g.size();
13    Generation g1 = new Generation(g);
14    for (Number i = new Zero(); i.lt(gSize); i=i.incBy2()){
15      Parents p = g.pickParents();
16      g1.add(p.tryCrossOver(probability))
17    }
18    if(g1.getDepth().lt(max)) announce GenAvailable(g1)
19  }
20  class Mutation {
21    Number probability; Number max;
22    init(...){
23      register(this)
24      // initialization elided (initializes fields above).
25    }
26    when GenAvailable do mutate;
27    void mutate(Generation g) {
28      Number gSize = g.size();
29      Generation g2 = new Generation(g);
30      for (Number i = new Zero(); i.lt(gSize); i=i.incBy2()){
31        Parents p = g.pickParents();
32        g2.add(p.tryMutation(probability))
33      }
34      if(g2.getDepth().lt(max)) announce GenAvailable(g2)
35    }
36  }
37  class Logger {
38    when GenAvailable do logit;
39    init(){ register(this) }
40    void logit(Generation g) { logGeneration(g) }
41  }
42  class Fittest {
43    Generation last;
44    when GenAvailable do check;
45    init(){ register(this) }
46    void check(Generation g) {
47      if(last == null) last = g;
48      else {
49        Fitness f1 = g.getFitness();
50        Fitness f2 = last.getFitness();
51        if(f1.average() > f2.average()) last = g
52      }
53    }
54  }

```

Figure 2. Pāṇini’s version of the Genetic algorithm

### 1.3 Contributions

Our language, Pāṇini, addresses these problems. The key idea behind Pāṇini’s design is to provide programmers with mechanisms to utilize prevalent idioms in modular program design. These mechanisms for modularity in turn automatically provide concurrency in a safe predictable manner. This paper discusses the notion of *asynchronous, typed events* in Pāṇini. An *asynchronous, typed event* exposes potential concurrency in programs which use behavioral design patterns for object-oriented languages, e.g. the observer pattern [20]. These patterns are widely adopted in software systems such as graphical user interface frameworks, middleware, databases, and Internet-scale distribution frameworks [20].

In Pāṇini, an *event type* is seen as a decoupling mechanism that is used to interface two sets of modules, so that they can be independent of each other. Below we briefly de-

scribe the syntax in the context of the genetic algorithm implementation in Pāṇini shown in Figure 2 (a more detailed description appears in Section 2). In the listing we have omitted initializations of classes for brevity. In this listing an example of an event type appears on lines 1–3, whose name is `GenAvailable` and that declares to make one context `g` of type `Generation` available.

Certain classes, to which we refer to as *subjects* from here onward, declaratively and explicitly announce events. The class `CrossOver` (lines 4–19) is an example of such a subject. This class contains a probability for the cross over operation and a maximum depth at which the algorithm will quit producing offspring. The method `cross` for this class computes the new generation based on the current generation (lines 11–19). After the `cross` method creates a new generation, it *announces* an event of type `GenAvailable` (line 18) denoted by code `announce GenAvailable(g1)`.

Another set of classes, which we refer to as *observers* from here onward, can provide methods, called *handlers* that are invoked (implicitly and *potentially concurrently*) when events are announced. The listing in Figure 2 has several examples of observers: `CrossOver`, `Mutation`, `Logger` and `Fittest`. A class can act as both subject and observer. For example, the classes `CrossOver` and `Mutation` are both subjects and observers for events of type `GenAvailable`.

In Pāṇini classes statically express (potential) interest in an event by providing a *binding declaration*. For example, the `Mutate` concern (lines 20–35) wants to randomly mutate some of the population after it is created. So in the implementation of class `Mutation` there is a binding declaration (line 26) that says to run the method `mutate` (lines 27–35) when events of type `GenAvailable` are announced.

At runtime these interests in events can be made concrete using the *register* expressions. The class `Mutation` has a method on lines 22–25 that when called registers the current instance `this` to listen for events. After registration, when any event of type `GenAvailable` is announced the method `mutate` (lines 27–35) will run with the registered instance `this` as the receiver object.

Similarly, the method `logit` (line 39) will log each generation when events of type `GenAvailable` are announced. Finally, method `check` in class `Fittest` (lines 41–51) will determine the better fitness between the announced generation and the previously optimal generation when events of type `GenAvailable` are announced.

**Benefits of Pāṇini’s Version of Genetic Algorithm.** At a quick glance, we can see from the shading that the four remaining concerns are no longer tangled and they are separated into individual modules. This separation not only makes reasoning about their behavior simple but also allows us to expose potential concurrency between them.

Furthermore, the concurrency concern has been removed entirely since the implementation of Pāṇini encapsulates all the code for managing the concurrency. By not requiring users to write this code, Pāṇini avoids any threat of incorrect or non-deterministic concurrency thus easing the burden on the programmer. This allows the programmer to focus on creating a good, maintainable modular design.

Finally, additional concurrency between these four modules is now automatically exposed. Thus, Pāṇini reconciles modular program design and concurrent program design.

### Advantages of Pāṇini’s Design over Related Ideas.

Pāṇini is most similar to our previous work on Ptolemy [37], but Pāṇini’s event types also have concurrency advantages.

It is also similar to implicit invocation (II) languages [13, 31] that also see *events* as a decoupling mechanism. The advantage of using Pāṇini over an II language is that asynchronous, typed events in Pāṇini allow developers to take advantage of the decoupling of subjects and observers to expose potential concurrency between their execution.

These events also relieve programmers from the burden of explicitly creating and maintaining threads, managing locks, and shared memory. Thus Pāṇini avoids the burden of reasoning about the usage of locks, which has several benefits. First, incorrect use of locks may have safety problems. Second, locks may degrade performance since acquiring and releasing a lock has overhead. Third, threads are cooperatively managed by the language runtime, thus thrashing due to excessive threading is avoided. These benefits make Pāṇini an interesting point in the design space of concurrent languages.

In summary, this work makes the following contributions:

1. Pāṇini’s language design with a simple and flexible implicit concurrency model such that Pāṇini programs are
  - free of data races,
  - free of deadlocks, and
  - have a guaranteed sequential semantics;
2. a precise operational semantics and type system for Pāṇini’s novel constructs;
3. an efficient implementation of Pāṇini’s design as an extension of the JastAdd compiler [14] that relies on:
  - a sound algorithm for finding inter-handler dependence at registration time to maximize concurrency,
  - a simple and efficient algorithm for scheduling concurrent tasks that builds on the fork/join framework [28];
4. a detailed analysis of Pāṇini and closely related ideas;
5. and, an empirical performance analysis of Pāṇini implementations using canonical concurrency examples.

### 1.4 Organization of this Paper

We have developed a small-step operational semantics, a type system and proved its soundness. Their details can be found in Appendix A (on page 15) and Appendix B (on

page 17). The next section describes the design and implementation of the Pāṇini language in detail. So a theoretical roadmap might be to read the next section followed by Appendix A and Appendix B.

Section 3 gives more examples in Pāṇini and describes our performance evaluation and experimental results. So an alternative roadmap might be to read the next two sections.

Finally, Section 4 surveys related work, and Section 5 describes future directions and concludes.

## 2. Pāṇini’s Design

*Pāṇini, fl. c.400 BC,*

*Indian grammarian, known for his formulation of the Sanskrit grammar rules, the earliest work on linguistics.*

In this section, we describe Pāṇini’s design. Pāṇini’s design builds on our previous work on the Ptolemy [37] and Eos [39] languages as well as implicitly parallel languages such as Jade [40]. Pāṇini achieves concurrent speedup by executing handler tasks (that do not conflict) concurrently. The novel features of Pāṇini are found in its concurrency model, event model and type system.

### 2.1 Pāṇini’s Abstract Syntax

```

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

```

**Figure 3.** Pāṇini’s abstract syntax, based on [10, 37].

Pāṇini features new mechanisms for declaring events and for announcing these events. These features are inspired by implicit invocation (II) languages such as Rapide [13] and concurrent languages such as Erlang [4]. The technical presentation of Pāṇini builds upon previous object-oriented calculi [10, 11, 19, 23, 37]. The object-oriented part of Pāṇini has classes, objects, inheritance, and subtyping, but it does not have **super**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods.

The abstract syntax is shown in Figure 3. A Pāṇini program consists of a sequence of declarations followed by an expression, which can be thought of as the body of a “main” method. In this syntax, the novel features are: event type declarations (**event**), event announcement expressions (**announce**), and handler registration expressions (**register**). Since Pāṇini 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. Examples of the syntax can be seen in Figure 2. This example is described thoroughly in Section 1.3.

**Top-level Declarations.** The two top-level declaration forms, classes and event type declarations, may not be nested. A class has exactly one superclass and may declare several fields, methods, and bindings. An event type (**event**) declaration has a name ( $p$ ), and zero or more context variable declarations ( $form^*$ ). These context declarations specify the types and names of reflective information exposed by conforming events. An example is given in Figure 2 on lines 1-3 where **event** GenAvailable has one context variable Generation  $g$  which denotes the generation which is now available. The intention of this event type declaration is to provide a named abstraction for a set of events that result from a generation being ready.

Like Eos [38] classes in Pāṇini may also contain binding declarations. A binding declaration mainly consists of two parts: an event type name and a method name. For example, in Figure 2 on line 10 the class CrossOver declares a binding such that the **cross** method is invoked whenever an **event** of type GenAvailable is announced. This method may run asynchronously with other handler methods.

### 2.2 Pāṇini’s Expressions

**Registration.** Like II languages, a module in Pāṇini can express interest in events, e.g. to implement the observer design pattern [20]. Just like II languages, where one has to write an expression for registering a handler with each event in a set, and similar to Ptolemy [37], such modules are defined using a binding inside a class declaration. Examples are shown on lines 7, 23, 38 and 44 in Figure 2.

**Object-oriented Expressions.** The formally specified version of Pāṇini is an expression language, thus the syntax for expressions includes several standard OO expressions [10, 11, 37]. These OO expressions include construction of an object (**new**  $c()$ ), variable dereference ( $var$ , including **this**), field dereference ( $e.f$ ), **null**, cast (**cast**  $t e$ ), assignment to a field ( $e_1.f = e_2$ ), a definition block ( $t var = e_1; e_2$ ), and sequencing ( $e_1; e_2$ ). Their semantics and typings are fairly standard [10, 11]. In the examples, we use conditionals of the form **if** ( $e == \mathbf{null}$ ) {  $e$  } **else** {  $e$  } and **for** ( $e; e; e$ ) {  $e$  }, which has standard desugaring.

**Concurrency in Pāṇini.** The **announce** expression enables concurrency in Pāṇini. The expression **announce**  $p (e^*) ; e$  signals an event of type  $p$ , which may run any handler bodies that are applicable to  $p$  *asynchronously*, and waits for the handlers to finish. In Figure 2 the body of the **cross** method contains an **announce** expression on line 18. On evaluation of the announce expression, Pāṇini first looks for

any applicable handlers. Here, the handlers `CrossOver`, `Mutation`, `Logger`, and `Fittest`, are declared to handle the events of type `GenAvailable`. Such handlers may run concurrently, depending on whether they interfere with each other.<sup>2</sup> The type system of Pāṇini computes the potential effects of the handlers.<sup>3</sup> Pāṇini will update the handlers’ potential effects when they register and decide which handlers conflict between each other. Pāṇini then schedules the handlers to maximize concurrency while maintaining the sequential semantics. The evaluation of the announce expression then continues with evaluating the sequence on line 18, which returns from the method. The announcement of the event allows for potential concurrent execution of the bodies of the `cross` (lines 11–19), `mutate` (lines 27–35), `logit` (line 38), and `check` (lines 45–51) methods.

The announce expression also binds values to the event type declaration’s context variables. For example, when announcing event `GenAvailable` on line 18, `g1` is bound to the context variable `g` on line 2. This binding makes the new generation available in the context variable `g`, which is needed by the context declared for the event type `GenAvailable`.

### 2.3 Pāṇini’s Registration Algorithm

**Handlers’ abstract read/write set.** To detect the dependency between any two handlers, two sets, namely the read and write set, are needed. The read set is a set of tuples of fields of classes that handler may read. Similarly, the write set stores tuples about fields that could be modified. Suppose we have two handlers,  $h_1$  and  $h_2$ . For a handler  $h_2$  to depend on  $h_1$ , there are two conditions: 1) handler  $h_1$  must register earlier than  $h_2$  in the program expression; 2)  $h_2$ ’s read set must conflict with  $h_1$ ’s write set (read after write [22]), or  $h_2$ ’s write set conflicts with either the read set of  $h_1$  (write after read [22]) or the write set of  $h_1$  (write after write [22]). That is because, in the sequential semantics,  $h_2$  should view the changes by  $h_1$ , while  $h_2$ ’s changes are invisible to  $h_1$ , neither should the changes of  $h_2$  be overwritten by the changes of  $h_1$ .

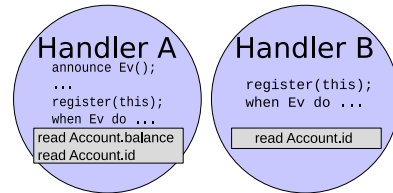
In Figure 4, handler  $A$  reads the field `balance` of the class `Account` and handler  $C$  may write to the field `balance`. Since handler  $A$  registers earlier than handler  $C$ , handler  $C$  depends on handler  $A$ , as discussed above.

Notice that a subscriber  $o$  could also be a publisher for an event, say  $p$ . Then the read/write set of  $o$  could be enlarged over time, because new handlers for  $p$  may register later and the effects of these new handlers should propagate to  $o$ . Pāṇini does these updates automatically when new handlers register for a certain event. For example, in Figure 5 notice that handler  $A$  may announce event type `Ev`, thus af-



**Figure 4.** Assume there are three handlers for the event type `Ev` in the program. At this point, none have registered yet. Handler  $A$  will register first, then handler  $B$ . Finally handler  $C$  will register last. The effect set of a handler is put inside the grey rectangle.

ter handler  $B$  registers, the effect set of handler  $A$  becomes the union of effect sets of handlers  $A$  and  $B$ . Finally, in Figure 6, the effect set of handler  $A$  becomes the union of effect sets all the three handlers.



**Figure 5.** After handler  $A$  and handler  $B$  have registered.



**Figure 6.** All the three handlers have registered.

**Event’s Subject list.** Subjects are formed into a list for an event. Thus, when a handler registers, its changes could be passed to these subscribers, and these subscribers merge the changes and recursively pass changes to other events when necessary. This continues until a fix point is reached (that is, no more effects are added to the publishers).

**Handlers’ Hierarchy.** Pāṇini groups handlers into hierarchies, based on handler dependencies. In the first level, none of the handlers have a dependency on any other handlers, while any handler in the second level depends on a subset of the handlers in the first level and no other handlers. For example, in Figure 7, handler  $C$  depends on handler  $A$  (this is discussed in more detail next). Similarly handlers in the third level may depend on handlers in the first two levels, but no handlers in any other level. It is possible that the effects of one handler will become larger (mentioned in the previous paragraph) and in response to this, Pāṇini will reorder the hierarchy dynamically.

<sup>2</sup> This is similar with Jade [40], where the implementation tries to discover concurrency. But unlike Jade, Pāṇini does not require programmers to provide any information.

<sup>3</sup> The type system is discussed in Appendix A.

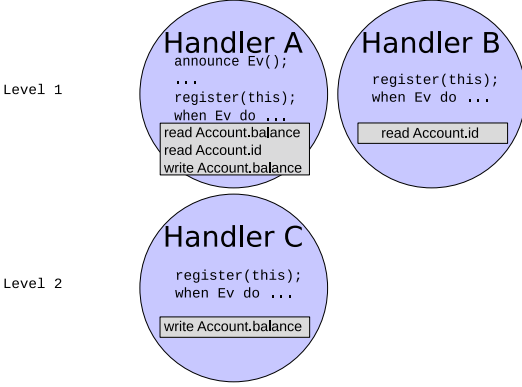


Figure 7. The handlers’ hierarchy for the event type *Ev*.

**Event registration.** When an event handler, say *h*, registers, we first propagate its effects to the publishers in the publishers list, then the dependencies between *h* and the previous registered handlers are computed based on the read / write set. After dependencies are calculated, the handler is put into a proper level of the hierarchy. In Figure 7, since, handler *A* may announce event type *Ev*, the effect sets of handler *B* and handler *C* are propagated to handler *A* (as a publisher). Because, handler *B* does not depend on handler *A* (notice that read effects of the same field have no conflict), it is put in the first level. Since handler *C* depends on handler *A*, it is put in the second level.

#### Event announcement and task scheduling algorithm.

When a publisher fires an event, Pānini executes the handlers in the first level concurrently. After all the handlers in this level are done, handlers in the next level are released and run in parallel until all the handlers are finished. For example, in Figure 7 since handlers *A* and *B* are both in the first level, they will run in parallel. Once they are completed, handler *C* will run.

The computation of the dependency and the effect propagation is done when handlers register, based on the assumption that in a program, the number of announcements considerably outweighs the number of registrations. Therefore, the overhead of effects manipulation is amortized. More details can be found in Appendix A and Appendix B.

## 2.4 Properties of Pānini’s Design

In this subsection, we study the key properties of Pānini’s design. We show that our language design has the following desirable properties. Well-typed Pānini programs do not get stuck and are free of races and deadlocks. The proof of this uses a standard preservation and progress argument [48] and is presented in Appendix C. The key novelty in the proof is the observation is that in Panini programs, there are not any data races between handlers. This is done mainly by the type system, as well as the scheduling algorithm during the event announcement, mentioned in 2.3.

## 2.5 Pānini’s Runtime System and Compiler

We designed an extension of Java to have asynchronous, typed events and implemented a compiler for this extension using the JastAddJ extensible compiler system [14]. As its backend, Pānini’s runtime system uses the fork/join framework [28]. This framework uses the work stealing algorithm [7] and works well for recursive work algorithms. We observed that handlers usually also act as subjects and recursively announce events, thus Pānini was built based on this framework. When an event is announced by a publisher, all handlers that are applicable are wrapped and put into the fork/join framework and may execute concurrently. Below we describe key parts of our implementation strategy.

```

1 public interface GenAvailable {
2   public Generation g();

3
4   public interface EventHandler extends IEventHandler {
5     public void ChangedHandle(Generation g); }

6
7   public interface EventPublisher extends IEventPublisher {

8
9   public class EventFrame implements GenAvailable {
10    public static void register(IEventHandler handler) {
11     /* 1. check whether this has register before,
12        if yes return ( no duplicate registration )
13        if no goto 2.
14        2. manipulate the effects of the handler
15        3. insert it into the handler hierarchy
16     */ }
17    public static void announce(GenAvailable ev) {
18     /* for( EventHandler eh:... )
19        // iterate all the handlers registered in the event
20        /* wrapping the handlers using GenAvailableTask */
21        PaniniTask.coInvoke(tasks);
22        // tasks is of type GenAvailableTask
23     */ }
24    /* other helper methods elided */
25   }

26
27   public static class GenAvailableTask extends PaniniTask {
28     /* public run(){ execute the handler method } */ }
29 }

```

Figure 8. An event type is translated into an interface with the same name (GenAvailable in the example).

**Event type.** An event type declaration is transformed into an interface (an example is shown in Figure 8). A getter method is generated for each context variable of the event (Generation *g* in line 2 in the example) so that the handlers can use this method to access the context variables. Two interfaces, namely *EventHandler* (lines 4–5) and *EventPublisher* (line 7), are to be used by an inner class *EventFrame* (lines 9–25), which hosts the register and announce methods for that event. Any class that has a binding declaration is instrumented to implement the *EventHandler* interface, while any class that may announce is instrumented to implement the *EventPublisher* interface.

**Event announcement.** When a subject announces an event, the announce method (line 17 in Figure 8) will be called. This method iterates over the handlers and executes all non-conflicting handlers level by level in the hierarchy

as discussed in 2.3. The class `EventFrame` uses a helper class (here `GenAvailableTask` on line 27), to wrap the handlers (if any) before submitting them for execution.

**Handler registration.** A register method is added to every class that has event bindings. First this method computes the effects of the handler. Next, this method registers to the named events in the class by calling the register method (line 10 in Figure 8). The register method in an event frame will first check that the current registering handler is in the handler hierarchy already to ensure no duplicate registration. Then the effects of the newly registered handler are compared again to other previously registered handlers to calculate the dependence set of this handler (discussed in 2.3). Finally, the handler is put into the handler hierarchy.

### 3. Evaluation

*In union there is strength. – Aesop*

This section describes evaluation of design and performance benefits of Pāṇini’s design.

#### 3.1 Analysis of Modularity and Concurrency Synergy

Pāṇini’s design can be seen as promoting both modularity and concurrency goals for a class of requirements that are typically modularized using behavioral design patterns [20].

The goal of this section is to analyze “if a program is modularized using Pāṇini’s features does that also expose potential concurrency in its execution?”

We have already presented one such case in Section 1, where modularization of various concerns in the implementation of a genetic algorithm exposed potential concurrency between these concerns. To further assess Pāṇini’s ability to achieve a synergy between modularity and concurrency goals, we have implemented several representative examples and they worked out beautifully. In the rest of this section, we present three examples. In these examples, we use Java extended with Pāṇini, which is supported by our compiler.

##### 3.1.1 Concurrency in Compiler Implementations

In the art of writing compilers, performance often has higher priority than modularity. Compiler designers employ all kinds of techniques to optimize their compilers. For example, merging transformation passes which perform different transformation tasks in the same traversal, is a common practice in writing multi-pass compilers. However, the implementation of this technique usually suffers the problem of code-tangling: implementations of different concerns (i.e., transformation tasks) are all mixed together.

Figure 9 illustrates this via snippets from an abstract syntax tree (AST). It shows concerns for method declarations, expressions, and two concrete expressions: a sequence expression (*e.e*) and a field get expression (*e.f*). As an example compiler pass, we show computation of effects for these AST nodes. The effect computation concern is scattered and

```

1 class MethodDecl extends ASTNode {
2   Expression body; // the expression body of the method
3   /* other fields and method elided */
4   Effect computeEffect() {
5     return body.computeEffect(); }
6 }
7 class Expression extends ASTNode {
8   Effect computeEffect();
9 class Sequence extends Expression {
10  Expression left; Expression right;
11  Effect computeEffect() {
12    Effect effect = left.computeEffect();
13    effect.add( right.getEffect() );
14    return effect; }
15 }
16 class FieldGet extends Expression {
17  Expression left; /* other fields elided */
18  Effect computeEffect() {
19    Effect effect = left.computeEffect();
20    effect.add( new ReadField() );
21    return effect; }
22 }

```

Figure 9. Snippets of an AST with an Effects System

tangled with the AST nodes. This is a common problem in compiler design where the abstract syntax tree hierarchy imposes a modularization based on language features whereas compiler developers may also want another modularization based on passes, e.g. type checking, error reporting, code generation, etc [14]. The visitor design pattern solves this problem to a certain extent but it has other problems [14].

```

1 event MethodVisited { MethodDecl md; }
2 event SequenceVisited { Sequence seq; }
3 event FieldGetVisited { FieldGet fg; }
4 class MethodDecl extends ASTNode {
5   Expression body; // the expression body of the method
6   /* other fields and method elided */
7   void visit() {
8     announce MethodVisited(this);
9     body.visit(); }
10 }
11 class Expression extends ASTNode { void visit() { } }
12 class Sequence extends Expression {
13   Expression left; Expression right;
14   /* other fields and method elided */
15   void visit() {
16     announce SequenceVisited(this);
17     left.visit(); right.visit(); }
18 }
19 class FieldGet extends Expression {
20   Expression left; /* other fields and method elided */
21   void visit() {
22     announce FieldGetVisited(this);
23     left.visit(); }
24 }
25 class ComputeEffect {
26   ComputeEffect() { register(this); h = new HashTable(); }
27   MethodDecl m; HashTable h;
28   when MethodVisited do start;
29   void start( MethodDecl md ) {
30     this.m = md;
31     h.add( m, new EffectSet() );
32   }
33   when FieldGetVisited do add;
34   void add( FieldGet fg ) {
35     h.get(m).add( new ReadField() );
36 } }

```

Figure 10. Pāṇini’s version of visiting an abstract syntax tree.

Pāṇini handles this modularization problem readily as shown in Figure 10. In this implementation, we introduce a method `visit` in each AST node. This method recursively visits the children of the node. At the same time, it announces events corresponding to the AST node. For example, a method declaration announces an event of type `MethodVisited` declared on line 1 and announced on line 8. Similarly, the AST node sequence expression and field get expression announce events of type `SequenceVisited` and `FieldGetVisited` on lines 16 and 22 respectively.

The implementation of effect computation is contained in the class `ComputeEffect`. This class has two binding declarations that say to run the method `start` when an event of type `MethodVisited` is announced and add when an event of type `FieldGetVisited` is announced. The constructor for this class registers itself to receive event announcements and initializes a hashtable to store effects per method. The method `add` inserts a read effect in this hashtable corresponding to the entry of the current method.

This Pāṇini program manifests a few design advantages. First AST implementation is completely separated from effect analysis. Also, unlike the visitor pattern, the `ComputeEffect` class need not implement a default functionality for all AST nodes. Furthermore, other passes such as type checking, error reporting, code generation, etc can also reuse the events declared for computing effects.

Last but not least, in Pāṇini, the effect computation (by the class `ComputeEffect`) could be processed in parallel with other compiler passes, like type checking. In case a compiler pass does transformation of AST nodes, Pāṇini’s type system will detect this as interference and automatically generate an order of their execution that would be equivalent to sequential execution. Thus, for this example Pāṇini shows that it can reconcile the modularity and concurrency goals such that modular design of compilers also improves their performance on multi-core processors.

### 3.1.2 Modular and Concurrent Image Processing

This example is adapted from and inspired by the ImageJ image processing toolkit [24]. For simplicity, assume that this library uses a class `List` and `Hashtable` similar to the classes in the `java.util` package. We have also omitted the irrelevant initializations of these classes. The class `Image` (lines 27-33) maintains a list of pixels. The method `set` for this class (lines 29–33) sets the value of a pixel at a given location to the specified integer value.

An example requirement for such a collection could be to signal changes of elements as an event. Other components may be interested in such events, e.g. for implementing incremental functionalities which rely on analyzing the increments. One such requirement for a list of pixels is to incrementally compute the Nonparametric Histogram-Based Thresholding [1, 21]. The threshold functionality may not be useful for all applications that use the image class, thus it would be sensible to keep its implementation separate

```

1 event Changed{
2   Image pic;
3 }
4 class Percentile extends Object{
5   when Changed do compute;
6   Hashtable h; /* Other fields omitted */
7   Percentile(){ register(this); h = new Hashtable(); }
8   void compute(Image pic){
9     /* Computationally intensive code for
10    computing percentile based threshold*/
11    h.add(pic, threshold);
12  }
13 }
14 class GlasbeyThreshold extends Object{
15   when Changed do compute;
16   Hashtable values;
17   GlasbeyThreshold init(){
18     register(this)
19   }
20   void compute(Image pic){
21     /* Computationally intensive code for
22    computing another type of threshold */
23    values.put(pic, threshold)
24  }}
25 class Image extends Object{
26   List pixels;
27   Image set(Integer i, Integer v){
28     pixels.setAt(i, v);
29     announce Changed(this);
30     this
31  }}

```

Figure 11. An Image and Threshold Computation in Pāṇini.

from the image class to maximize reuse of the image class. Pāṇini’s implementation allows the threshold computation concerns to remain independent of the image concerns, while allowing their concurrent execution.

### 3.1.3 Overlapping Communication with Computation via Modularization of Concerns

Our next example presents a simple application for planning a trip. Planning requires finding available flights on the departure and return dates as well as a hotel and rental car for the duration of the trip. To find each of these items the program must communicate with services provided by other providers and each computation can be run independently.

In this example the context variable `tripData` is used to both provide the handlers with information and to give the handlers a place to store their results. As with the above example, initialization functions are elided. The `FilterCars` handler is also elided. The lists are filtered with an event since filtering the lists can be done independently and concurrently. In this example as well Pāṇini’s design shows the potential of reconciling modularity goals with concurrency goals. When an event of type `PlanTrip` is announced each of the three handlers can execute concurrently.

## 3.2 Performance Evaluation

The goal of this section is to analyze “how well do the Pāṇini programs perform compared to a hand-tuned concurrent implementation of equivalent functionality?”

We first describe our experimental setup and then analyze speedup realized by Pāṇini’s implementation as well as the overheads.

```

52 event PlanTrip{
53   TripData td;
54 }
55 class CheckAirline{
56   init(...){
57     register(this)
58   }
59   when PlanTrip do checkFlights;
60   //Find all the available flights during the trip
61   void checkFlights(TripData td){
62     AvailableFlights flights1
63     = getAirline1Flights(td.depart,td.arrive);
64     AvailableFlights flights2
65     = getAirline2Flights(td.depart, td.arrive);
66     AvailableFlights flights3
67     = getAirline3Flights(td.depart, td.arrive);
68     //add the results to the tripData
69     td.setAvailableFlights(flights1, flights2, flights3)
70 }
71 class CheckHotel{
72   init(...){
73     register(this)
74   }
75   when PlanTrip do checkHotel;
76   //Search for available hotels.
77   void checkFlights(TripData td){
78     AvailableHotels hotels
79     = HotelsProvider.search(
80       td.depart, td.arrive, td.pricePref);
81     td.setAvailableHotels(hotels)
82 }
83 class CheckRentalCar{
84   init(...){
85     register(this)
86   }
87   when PlanTrip do checkCarRentals(TripData){
88     AvailableCars rental1
89     = getRentals("RentalAgency1",
90       td.depart, td.arrive, td.carPref);
91     AvailableCars rental2
92     = getRentals("RentalAgency2",
93       td.depart, td.arrive, td.carPrefs);
94     //filter the lists of rentable cars
95     announce FilterCars(rental1, td);
96     announce FilterCars(rental2, td);
97     td.addRentalChoices(rental1);
98     td.addRentalChoices(rental2)
99 }

```

Figure 12. Accessing service providers in handlers.

All experiments were performed on a system with a total of 12 cores (2x6-core AMD Opteron 2431 chips) running Fedora GNU/Linux.

### 3.2.1 Concurrency Benchmark Selection

To avoid bias in the performance measurement, we picked already implemented concurrent solutions of five computationally intensive kernels: Euler number, FFT, Fibonacci, integrate, and merge sort. Hand-tuned implementations of these kernels were already available [28].

Each program takes an input to vary the size of the workload (Euler: number of rows, FFT: size of matrix  $2^x$ , Fibonacci:  $x^{th}$  Fibonacci number, integrate: number of exponents, and merge sort: array size  $2^x$ ) For each example program, a sequential version was tested as well as concurrent versions ranging from 1 to 14 threads. Furthermore, three concurrent versions were tested:

1. an implementation using the fork/join framework [28],

2. a Pānini version with no conflict between handlers, and
3. a second Pānini’s implementation that was intentionally designed to have conflicts between handlers.

For example, calculating a Fibonacci number,  $fib(n)$ , is done by recursively calculating two subproblems,  $fib(n-1)$  and  $fib(n-2)$ . With the fork/join framework, each of these subproblems is done by a separate thread. When both of these threads are completed, the spawning thread adds them together. For Pānini, each of these subproblems is handled in separate handlers. In the case with no conflicts, these are the only two handlers. In the case with conflicts, a third handler takes the result of the two handlers for the subproblems and adds them together. In all cases, when the problem size is small enough, computation continues sequentially.

### 3.2.2 Speedup over Sequential Implementation

We now analyze the speedup of parallel programs achieved by the Pānini’s implementation as compared to the standard fork/join model [28].

Figure 13 shows a summary comparison of speedup between the three versions. In this figure, the average speedup across all five benchmarks was taken. For each program, large input sets were used (Euler: 39, FFT: 24, Fibonacci: 55, integrate: 7, and merge sort: 25 ). The line in the figure represents optimal speedup.

This figure shows that the speedups between the three styles are comparable. Speedups for fork/join and Pānini without conflicts are nearly the same.

A statistical analysis shows that for all benchmarks, we do not see a statistically significant difference ( $p < 0.05$ ) between fork/join and Pānini with no conflicts.

From the figure, we can also see that Pānini with conflicts has slightly lower speedup than both fork/join and Pānini with conflicts, however, this decrease is rather small (average 6.5% decrease from fork/join). We found a statistically significant difference ( $p < 0.05$ ) between fork/join and Pānini with conflicts. Note that since we are using a machine with 12 cores, performance levels off at 12 threads.

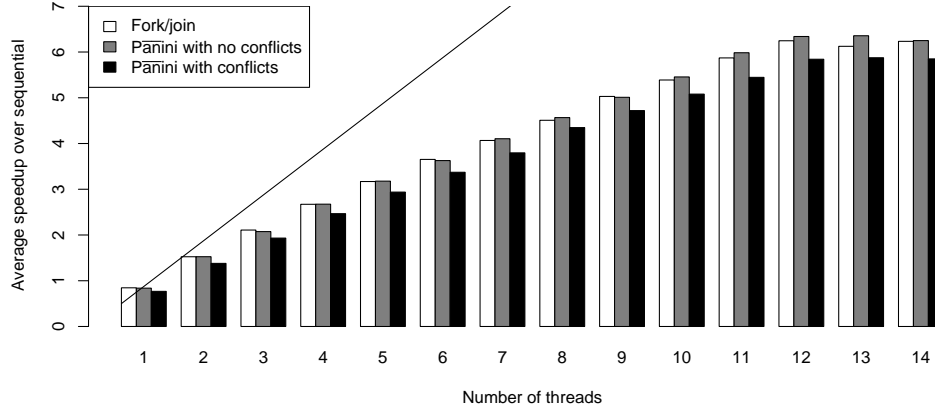
### 3.2.3 Overhead over the Sequential Implementation

We also measured the overhead involved with Pānini as compared to the standard fork/join model.

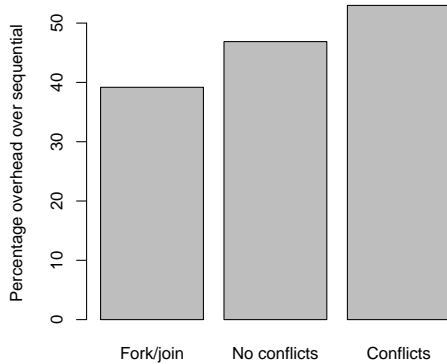
We first consider the average overhead across all benchmarks as shown in Figure 14. Overhead is calculated by determining the increase in runtime from the sequential version to the *concurrent version with a single thread*. For this figure, we use large input sizes.

This figure shows us that while Pānini increases the overhead over fork/join, it is not a prohibitive amount. For example, from fork/join to Pānini with no conflicts, we only see a 7.7% increase in overhead. Similarly, adding conflicts to Pānini only incurs an additional 6.1% overhead.

Figure 15 shows a summary comparison of overhead as program input size changes. In this figure, the overhead for



**Figure 13.** Average speedup compared to sequential version across all benchmarks for varying number of threads. Line represents perfect scaling. Shows that Pāṇini’s implementation scale similar to hand-written fork/join implementation.



**Figure 14.** Average overhead compared to sequential version across all benchmarks for each technique.

the Fibonacci program is shown with a variety of input sizes. Again, overhead is calculated by determining the increase in runtime from the sequential version to the concurrent version with a single thread.

This figure shows that as input size increases, overhead decreases. In this case, overhead decreases to as low as 5.5% additional overhead for Pāṇini with no conflicts. Pāṇini with conflicts only incurs an additional 1.2% overhead for larger input sizes. Each of the differences in overhead (fork/join vs Pāṇini without conflicts, fork/join vs Pāṇini with conflicts, and Pāṇini with vs Pāṇini without conflicts) was statistically significant ( $p < 0.05$ ) across all benchmarks.

### 3.3 Summary of Results

In summary, Pāṇini shows speedups which scale as well as the standard fork/join model. Even though Pāṇini has a higher overhead than fork/join, Pāṇini performs nearly as well as the fork/join model in terms of speedup for nearly all cases. This is all achieved without requiring explicit concurrency and while encouraging good modular design and ensuring that programs are free of deadlocks and have sequential semantics.

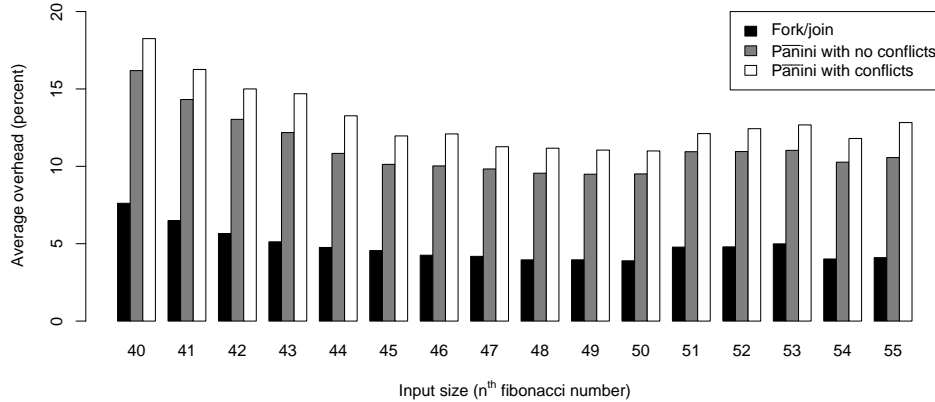
## 4. Related Work

*conciliate, mediate, harmonize, settle, accommodate, reunite – entry for “reconcile” in Roget’s II*

Like Jade [40], Pāṇini is an implicit concurrency language. Programmers in Jade supply information about the effect of tasks so that the implementation may discover concurrency. Pāṇini is different in that it relies on its type system to maximize its effects and removes the burden on the programmer to supply these effects by hand. Pāṇini also removes any errors which could be introduced by incorrect specification of effects. This is different from Grace [6] which is an explicit threading language. Grace executes threads speculatively. If a conflict is detected, it rolls back the changes. If no races are detected it commits the changes. Pāṇini detects conflict when handlers register.

Like X10 [9], Pāṇini does not feature any construct for explicit locking. However, X10 is an explicit concurrency language and it uses *atomic blocks* for lock-free synchronization and uses the concept *clocks* as synchronization between *activities*. The Task Parallel Library (TPL) [29], wraps computation into tasks and uses thread stealing as the underlying implementation. This is similar to Pāṇini’s runtime, but programmers in TPL have to explicitly account for races, whereas Pāṇini’s effect system automatically avoids all races.

Similar with the effect sets of Pāṇini, deterministic parallel Java (DPL) [8] uses effect sets to provide deterministic semantics for programs. DPL eliminates races by its type system which relies on read/write information for all methods. For DPL, this read/write information is explicitly provided by the programmer. This is unlike Pāṇini which relieves programmers of the burden of writing any specifications for the effects of methods. Like Pāṇini, DPL’s runtime is also built upon the Fork/Join [28] framework. DPL provides programmers with two concurrent constructs to parallelize their programs. This is unlike Pāṇini, which does not require programmers to construct explicitly parallel pro-



**Figure 15.** Average overhead for Fibonacci benchmark for varying input size and each scheduling strategy.

grams. Instead, Pānini promotes the goal of writing programs with good modular designs.

Events have a long history in both the software design [12, 18, 30, 31, 45] and distributed systems communities [17, 25, 32]. Pānini’s notion of asynchronous, typed events build on these notions, in particular recent work in programming languages focusing on event-driven design [15, 16, 37]. In software design, events and implicit invocation have been seen as a decoupling mechanism for modules [31, 45], whereas in distributed systems, events are seen as a mechanism of decoupling component execution for location transparent deployment and extensibility [32]. New to Pānini’s design and its philosophy is the unification of design and execution decoupling.

Pānini’s design is also not the first to promote implicit concurrency. For example, in POOL [3], ABCL [49], Concurrent Smalltalk [50] and BETA [44], objects implicitly execute in the context of a local process. This is different from Pānini where only handler instances are run implicitly and concurrently. This allows smoother integration with mainstream programming languages such as Java. This also permits an easier integration of our event-based model with the thread-based explicit concurrency models as promoted by Li and Zdancewic [30]. In this work, we do not discuss the semantic issues with this integration, however.

Other recent work such as TaskJava [18] in the programming language design setting and Tame [25] and Tasks [12], in the distributed systems setting, have promoted similar integration with existing languages. Fischer *et al.* in their work on TaskJava [18] introduce `task`, similar to a handler in Pānini. An `asynchronous` method is marked with the keyword `async`, indicating that it could block and should be compiled into continuation-passing style. This method may use a primitive `wait` to express its interests in a set of events and this expression will block until one of them fires. Krohn *et al.* have promoted similar ideas in Tame [25] where they use a primitive `twait` to block on events. In both these approaches, running of the concurrent task is explicitly man-

aged by the programmer. In Pānini, however, handlers are implicitly spawned and managed by the language runtime. As a result, programmers are relieved of reasoning about locking and data race problems. 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.

Unlike Multilisp [41], which has the future construct, Pānini uses different expressions as synchronization points. Moreover, unlike Java’s current adoption of Futures, which is unsafe [47], heap access expressions in Pānini are sound. Furthermore, unlike previous work [35, 47], our implementation doesn’t require modifications to the virtual machine.

## 5. Conclusion and Future Work

*Faith is taking the first step even when you don’t see the whole staircase. – Martin Luther King Jr.*

Language features that promote concurrency in program design have become important [2, 5]. Explicit concurrency features such as threads are hard to reason about and building correct software systems in their presence is difficult [33, 43]. There have been several proposals for concurrent language features, but none unifies program design for modularity with program design for concurrency. In the design of Pānini, we pursue this goal. In an effort to do so, we have developed the notion of events that are especially helpful for programs where modules are decoupled using implicit-invocation design style [13, 31, 32, 45]. Event announcements provide implicit concurrency in program designs when events are signaled and consumed. We have tried out several examples, where Pānini improves both program design and potential available concurrency.

For future work, we would like to apply Pānini’s design to large projects and to evaluate whether the reconciliation

of modularity and concurrency goals that we saw in our examples is scalable to larger software projects.

## Acknowledgments

This work was supported in part by the NSF under grant CCF-08-46059. Comments and discussions with Robert Dyer, Steven M. Kautz, Gary T. Leavens were helpful.

## References

- [1] HistThresh toolbox for MATLAB. <http://www.cs.tut.fi/~ant/histthresh/>.
- [2] M. Abadi and G. Plotkin. A model of cooperative threads. In *the 36th Symposium on Principles of Programming Languages (POPL)*, pages 29–40, 2009.
- [3] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
- [4] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.
- [5] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
- [6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, New York, NY, USA, 2009. ACM.
- [7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP*, pages 207–216, 1995.
- [8] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2009. ACM.
- [9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.
- [10] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.
- [11] C. Clifton and G. T. Leavens. MiniMAO<sub>1</sub>: Investigating the semantics of proceed. *Science of Computer Programming*, 63(3):321–374, 2006.
- [12] R. Cunningham and E. Kohler. Tasks: language support for event-driven programming. In *Conference on Hot Topics in Operating Systems*, Berkeley, CA, USA, 2005.
- [13] David C. Luckham *et al.*. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, 1995.
- [14] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA*, pages 1–18, 2007.
- [15] P. Eugster. Type-Based Publish/Subscribe: Concepts and Experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.
- [16] P. Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *ECOOP*, pages 570–584, 2009.
- [17] P. T. Eugster, R. Guerraoui, and C. H. Damm. On Objects and Events. In *OOPSLA*, pages 254–269, 2001.
- [18] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *PEPM*, pages 134–143, 2007.
- [19] 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. 1999.
- [20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [21] C. A. Glasbey. An analysis of histogram-based thresholding algorithms. *CVGIP: Graphical Models and Image Processing*, 55(6):532 – 537, 1993.
- [22] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
- [23] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, pages 132–146, 1999.
- [24] Image Processing and Analysis in Java. ImageJ. <http://rsbweb.nih.gov/ij/>.
- [25] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX*, 2007.
- [26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.
- [27] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [28] D. Lea. A Java Fork/Join Framework. In *Java Grand*, pages 36–43, 2000.
- [29] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 227–242, New York, NY, USA, 2009. ACM.
- [30] 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.
- [31] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, 1993.

- [32] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. In *SOSP*, pages 58–68, 1993.
- [33] J. Ousterhout. Why threads are a bad idea (for most purposes). In *ATEC*, January 1996.
- [34] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60, New York, NY, USA, 2008. ACM.
- [35] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *OOPSLA*, pages 206–223, 2004.
- [36] H. Rajan and G. T. Leavens. Quantified, typed events for improved separation of concerns. Technical Report 07-14, Iowa State University, Department of Computer Science, July 2007. In submission.
- [37] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.
- [38] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *the European software engineering conference and international symposium on Foundations of software engineering (ESEC/FSE)*, pages 297–306, 2003.
- [39] H. Rajan and K. Sullivan. Need for instance level aspect language with rich pointcut language. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.
- [40] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.
- [41] J. Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
- [42] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.
- [43] Saha, B. *et al.*. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.
- [44] B. Shriver and P. Wegner. Research directions in object-oriented programming, 1987.
- [45] K. J. Sullivan and D. Notkin. Reconciling Environment Integration and Component Independence. *SIGSOFT Software Engineering Notes*, 15(6):22–33, December 1990.
- [46] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
- [47] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2005.
- [48] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994.
- [49] A. Yonezawa. ABCL: An object-oriented concurrent system, 1990.
- [50] A. Yonezawa and M. Tokoro. Object-oriented concurrent programming, 1990.

## A. Pānini's Type and Effect System

Type checking uses the attributes defined in Figure 16. Compared to Ptolemy [37] new to Pānini is its effect system. For example, the type attributes for expressions are represented as  $(t, \rho)$ , the type of the expression ( $t$ ) and its effect set ( $\rho$ ).

$\theta ::= \text{OK}$	“program/top-level decl/body types”
$\quad   (t_1 \times \dots \times t_n \rightarrow t, \rho) \text{ in } c$	“method types”
$\quad   (t, \rho)$	“expression types”
$\rho ::= \epsilon + \rho \mid \bullet$	“program effects”
$\epsilon ::= \text{read } c f$	“read effect”
$\quad   \text{write } c f$	“write effect”
$\quad   \text{ann } p$	“announce effect”
$\quad   \text{reg}$	“register effect”
$\quad   \text{create } c$	“new object effect”
$\pi, \Pi ::= \{I : \theta_I\}_{I \in K},$	“type environments”
$\quad \text{where } K \text{ is finite, } K \subseteq (\mathcal{L} \cup \{\mathbf{this}\}) \cup \mathcal{V}$	

**Figure 16.** Type and Effect Attributes.

The effects are used to compute the potential conflicts between handlers. These effects include: 1) read effect: a class and a field to be read; 2) write effect, content is similar to read effect; 3) announce effect: what event a certain expression may announce; 4) register effect and 5) new object. The interference between the effects is shown in Figure 17.

Effects	read	write	ann	reg	create
read	×	✓	×	✓	×
write	✓	✓	×	✓	×
ann	×	×	×	✓	×
reg	✓	✓	✓	✓	×
create	×	×	×	×	×

**Figure 17.** Effect Interference. ✓: conflicts, ×: no conflicts

Read effects will not interfere with each other. Write effects will conflict if either another read or write effect accesses the same field of the same object. Announce effects will interfere with register effects, because the order of registration affects the set of handlers run during announcement. Announce effect is also used later in the semantics because handlers could also act as publishers (refer to as handler/publisher) and announce events ( $e$ ). Pānini updates the effects of these handler/publisher(s) every time a handler registers the event  $e$ . Thus Pānini will get more accurate information about the effects of the handlers/publishers when scheduling and reduce false interferences. In Pānini, register effects will interfere with read/write effects as well, due to the fact that after a register, a handler could introduce unforeseen read and write effects and thus complicate the interference.

New object effect will not interfere with any other effect. The new object effect is used to reduce false interference. Certain variables are marked as **create** if type checking detects that these variables point to newly created objects. We observe that new objects are not the sources for interference for the following reasons:

1. if a newly created object does not escape from the handler the object can not be accessed by any other handlers, thus there will not be any race;

2. otherwise, assume that a newly created object ( $o_n$ ) escapes the handler ( $h_1$ ) and is referenced by another handler ( $h_2$ ), then the program will first have to change a field of another object (referred to as  $o_a$ ) to point to  $o_n$  to make it escape. On the other hand,  $h_2$  will have to read the field of  $o_a$ , which will be detected by Pānini and reported as an interference (because  $h_1$  changes the field of  $o_a$  and  $h_2$  reads the field of  $o_a$ ). That is to say, it will not be the newly created object that causes data races.

Thanks to this observation, Pānini could safely remove the read/write effect of any newly created object and thus reduce false interferences to a considerable extent.

The type checking rules are shown in Figures 18 and 20. The notation  $\nu' <: \nu$  means  $\nu'$  is a subtype of  $\nu$ . It is the reflexive-transitive closure of the declared subclass relationships [37]. We state the type checking rules using a fixed class table (list of declarations  $CT$ ) as in Clifton's work [10,11]. 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,  $\Pi$ , which is a finite partial mapping from locations  $loc$  or variable names  $var$  to a type.

<p>(T-PROGRAM)</p> $\frac{(\forall decl_i \in \overline{decl} :: \vdash decl_i : \text{OK})}{\overline{decl} \vdash e : (t, \rho)} \quad \text{OK}$	<p>(T-EVENT)</p> $\frac{(\forall (t_i \text{ var}_i) \in \overline{t \text{ var}_i} :: \text{isType}(t_i))}{\vdash \text{event } p \{ \overline{t \text{ var}_i} \} : \text{OK}}$
<p>(T-CLASS)</p> $\frac{\text{validF}(\overline{t \text{ f}}, d) \quad \text{isClass}(d) \quad (\forall \text{meth}_j \in \overline{\text{meth}} :: \vdash \text{meth}_j : (\theta_j, \rho_j) \text{ in } C) \quad (\forall b \in \overline{\text{binding}} :: \vdash b : \text{OK in } C)}{\vdash \text{class } c \text{ extends } d \{ \overline{t \text{ f}}; \overline{\text{meth}} \text{ binding} \} : \text{OK}}$	
<p>(T-METHOD)</p> $\frac{\text{isClass}(t) \quad (\forall i \in \{1..n\} :: \text{isClass}(t_i)) \quad \text{var}_1 : t_1, \dots, \text{var}_n : t_n, \mathbf{this} : c \vdash e : (u, \rho) \quad u <: t \quad \text{override}(m, c, (t_1 \times \dots \times t_n \rightarrow t, \rho))}{\vdash t m(t_1 \text{ var}_1, \dots, t_n \text{ var}_n) \{ e \} : (t_1 \times \dots \times t_n \rightarrow t, \rho) \text{ in } c}$	
<p>(T-BINDING)</p> $\frac{CT(p) = \text{event } p \{ t'_1 \text{ var}'_1, \dots, t'_m \text{ var}'_m \} \quad (c_1, t m(\overline{t \text{ var}}) \{ e \}, \rho) = \text{findMeth}(c, m) \quad \pi = \{ \text{var}'_1 : t'_1, \dots, \text{var}'_m : t'_m \} \quad (\forall t_i \text{ var}_i \in \overline{t \text{ var}} :: \pi(\text{var}_i) <: t_i)}{\vdash \text{when } p \text{ do } m : \text{OK in } c}$	

**Figure 18.** Type and Effect rules for declarations [10, 11, 37].

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  $\rho$ .

The (T-EVENT) rule says that an event declaration type checks, if all the types of all the fields are declared properly. The auxiliary function  $\text{isType}$  (shown in Figure 21), looks at the class table to check if a type has been defined or not.

Valid method overriding:

$$\begin{array}{c}
CT(c) = \mathbf{class} \ c \ \mathbf{extends} \ d \ \{ \mathit{field}^* \ \mathit{meth}_1 \dots \mathit{meth}_p \ \mathit{bind}_1 \dots \mathit{bind}_q \} \\
\#i \in \{1..p\} \cdot \mathit{meth}_i = t \ m(t_1 \ \mathit{var}_1, \dots, t_n \ \mathit{var}_n) \{ e \} \\
\mathit{override}(m, d, (t_1 \times \dots \times t_n \rightarrow t, \rho)) \\
\hline
\mathit{override}(m, c, t_1 \times \dots \times t_n \rightarrow t, \rho) \\
\hline
\mathit{methodType}(d, m) = (t_1 \times \dots \times t_n \rightarrow t, \rho') \quad \rho \subseteq \rho' \\
\mathit{override}(m, d, (t_1 \times \dots \times t_n \rightarrow t, \rho)) \\
\hline
\mathit{override}(m, \mathit{Object}, t_1 \times \dots \times t_n \rightarrow t, \rho)
\end{array}$$

**Figure 19.** Auxiliary functions used in type rules [10, 11].

$$\begin{array}{c}
\text{(T-NEW)} \quad \frac{\mathit{isClass}(c)}{\Pi \vdash \mathbf{new} \ c() : (c, \{\mathbf{create} \ c\})} \quad \text{(T-CAST)} \quad \frac{\mathit{isType}(t) \quad \Pi \vdash e : (u, \rho)}{\Pi \vdash \mathbf{cast} \ t \ e : (t, \rho)} \\
\text{(T-GET)} \quad \frac{\Pi \vdash e : (c, \rho) \quad \rho \neq \{\mathbf{create} \ c\} \quad \mathit{fieldsOf}(c)(f) = t}{\Pi \vdash e.f : (t, \rho \cup \{\mathbf{read} \ c \ f\})} \quad \text{(T-GET-LOCAL)} \quad \frac{\Pi \vdash e : (c, \{\mathbf{create} \ c\}) \quad \mathit{fieldsOf}(c)(f) = t}{\Pi \vdash e.f : (t, \{\})} \\
\text{(T-SEQUENCE)} \quad \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')} \quad \text{(T-YIELD)} \quad \frac{\Pi \vdash e : (t, \rho)}{\Pi \vdash \mathbf{yield} \ e : (t, \rho)} \\
\text{(T-VAR)} \quad \frac{\Pi(\mathit{var}) = (t, \rho)}{\Pi \vdash \mathit{var} : (t, \rho)} \quad \text{(T-DEFINE)} \quad \frac{\mathit{isType}(t) \quad \Pi \vdash e_1 : (t_1, \rho) \quad \Pi, \mathit{var} : (t, \rho) \vdash e_2 : (t_2, \rho') \quad t_1 <: t}{\Pi \vdash t \ \mathit{var} = e_1; e_2 : (t_2, \rho \cup \rho')} \\
\text{(T-NULL)} \quad \frac{\mathit{isClass}(c)}{\Pi \vdash \mathbf{null} : (c, \emptyset)} \quad \text{(T-REGISTER)} \quad \frac{\Pi \vdash e : (t, \rho) \quad \mathit{isClass}(t)}{\Pi \vdash \mathbf{register} \ (e) : (t, \rho \cup \{\mathbf{reg} \})} \\
\text{(T-SET)} \quad \frac{\Pi \vdash e : (c, \rho) \quad \rho \neq \{\mathbf{create} \ c\} \quad \mathit{fieldsOf}(c)(f) = t \quad \Pi \vdash e' : (t', \rho') \quad t' <: t}{\Pi \vdash e.f = e' : (t', \rho \cup \rho' \cup \{\mathbf{write} \ c \ f\})} \\
\text{(T-SET-LOCAL)} \quad \frac{\Pi \vdash e : (c, \{\mathbf{create} \ c\}) \quad \mathit{fieldsOf}(c)(f) = t \quad \Pi \vdash e' : (t', \rho') \quad t' <: t}{\Pi \vdash e.f = e' : (t', \rho \cup \rho')} \\
\text{(T-ANNOUNCE)} \quad \frac{CT(p) = \mathbf{event} \ p \ \{ t_1 \ \mathit{var}_1; \dots t_n \ \mathit{var}_n; \} \quad (\forall i \in \{1..n\} :: \Pi \vdash e_i : (u_i, \rho_i) \wedge u_i <: t_i) \quad \Pi \vdash e : (t, \rho)}{\Pi \vdash \mathbf{announce} \ p \ (e_1, \dots, e_n) ; e : (t, \{\mathbf{ann} \ p\} \cup \bigcup_{i=1}^n \rho_i \cup \rho)} \\
\text{(T-CALL)} \quad \frac{(c_1, t \ m \ (t_1 \ \mathit{var}_1, \dots, t_n \ \mathit{var}_n) \ \{ e_{n+1} \}, \rho) = \mathit{findMeth}(c_0, m) \quad c_0 <: c_1 \quad \Pi \vdash e_0 : (c_0, \rho_0) \quad (\forall i \in \{1..n\} :: \Pi \vdash e_i : (u_i, \rho_i) \wedge u_i <: t_i)}{\Pi \vdash e_0.m(e_1, \dots, e_n) : (t, \rho \cup \bigcup_{i=1}^n \rho_i \cup \rho_0)}
\end{array}$$

**Figure 20.** Type and Effect rules for expressions [10,11,37].

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 if all the following constraints are satisfied. First, the return type is a class type. Next, if all the parameters have their corresponding declared types, the body of the method has type *u* and effect  $\rho$ . Also *u* is a subtype of *t*. This rule also uses an auxiliary function *override*, defined in Figure 19. In addition to standard conditions, this function enforces that the effect of an overriding method is the subset of the effect of overridden method<sup>4</sup>.

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; and the named event type is declared.

The type rules for the expressions are shown in Figure 20. Most rules for typing expressions are straightforward.

The (T-NEW) rule says that a new expression has the type of the class being declared if the class *c* has been properly declared and has a single effect **create** to denote that this is a newly created object as mentioned previously to reduce false interferences.

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-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 (T-GET-LOCAL) rule is similar to the previous rule, except that Pāṇini knows that the object expression is pointing to a newly created object and thus the read effect is redundant and deleted.

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 properly.

The (T-NULL) rule says that the null expression will type check and has no effect.

The (T-REGISTER) rule says that a register expression has the same type as the object expression and the effects will be the effects of the object expression plus one register effect.

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*<sub>2</sub> is a subtype of the type of the field of the class. The effects will be the union of

<sup>4</sup>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.

the effects of its two subexpressions plus one **write** effect since this expression is to modify a field of an object.

The (T-SET-LOCAL) rule is similar to (T-SET) except that the type system can detect that it is changing a field of a new object thus the single **write** is not needed.

The (T-ANNOUNCE) rule says that an announcement expression type checks if the event was declared and the actual parameters are a subtype of the declared fields type in the event declaration. The entire expression has the type of the subsequent expression  $e$ . The effects of the announce expression will be the union of all the parameters' effects and the subsequent expression plus an 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` (not shown here) 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. The auxiliary function `findMeth` works similar to the `methodName` in Clifton's work [10, 11], except that here it also returns the effect set of the method.

Valid method overriding:

$$isType(t) = (t \in dom(CT) \wedge CT(t) = \mathbf{class} \ t \dots)$$

$$\begin{aligned} fieldsOf(c) &= \{t_i\} \cup fieldsOf(c') \\ \mathbf{where} \ CT(c) &= \mathbf{class} \ c \ \mathbf{extends} \ c' \{t_1 \ f_1; \dots t_n \ f_n; \dots\} \\ validF(\bar{t} \ \bar{f}, c) &= \forall i \in \{1..n\} :: isType(t_i) \wedge f_i \notin dom(fieldsOf(c)) \end{aligned}$$

**Figure 21.** Auxiliary functions used in type rules [36].

## B. Pāṇini's Operational Semantics

Here we give a small-step operational semantics for Pāṇini.

**Intermediate Expressions.** The expression semantics rely on four expressions that are not part of Pāṇini's surface syntax as shown in Figure 22. The  $loc$  expression represents locations in the store. Following Abadi and Plotkin [2], we use the **yield** expression to model concurrency. The **yield** expression allows other tasks to run. The rules and auxiliary functions all make implicit use of a (global) list:  $CT$ , the program's declarations. The `NullPointerException` and `ClassCastException` (shown in Figure 23) are two final states reached: 1) when trying to access a field or a method from a **null** pointer object or 2) an object that is not of subtype of the casting type.

**Domains.** The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 22.

A configuration consists of a task queue  $\psi$ , a global store  $\mu$ , and a global subscriber list  $\gamma$ . The store  $\mu$  is a mapping

### Added Syntax:

$$e ::= loc \mid \mathbf{yield} \ e \mid \mathbf{NullPointerException} \mid \mathbf{ClassCastException}$$

where  $loc \in \mathcal{L}$ , a set of locations

**Evaluation relation:**  $\hookrightarrow: \Sigma \rightarrow \Sigma$

### Domains:

$$\begin{aligned} \Sigma &::= \langle \psi, \mu, \gamma \rangle && \text{"Program Configurations"} \\ \psi &::= \langle e, \tau \rangle + \psi \mid \bullet && \text{"Task Configurations"} \\ \tau &::= \langle n, \{n_k\}_{k \in K} \rangle && \text{"Task Local Data"} \\ &\mathbf{where} \ n_k \in \mathcal{N} \ \mathbf{and} \ K \ \mathbf{is} \ \mathbf{finite} \\ \mu &::= \{loc \mapsto o\} + \mu \mid \bullet && \text{"Stores"} \\ v &::= \mathbf{null} \mid loc && \text{"Values"} \\ o &::= [c.F] && \text{"Object Records"} \\ F &::= \{f_k \mapsto v_k\}_{k \in K}, && \text{"Field Maps"} \\ &\mathbf{where} \ K \ \mathbf{is} \ \mathbf{finite} \\ \gamma &::= loc + \gamma \mid \bullet && \text{"Subscriber List"} \end{aligned}$$

### Evaluation contexts:

$$\begin{aligned} \mathbb{E} &::= - \mid \mathbb{E} . m(e \dots) \mid v . m(v \dots \mathbb{E} e \dots) \mid \mathbf{cast} \ t \ \mathbb{E} \mid \mathbb{E} . f \\ &\mid \mathbb{E} . f = e \mid v . f = \mathbb{E} \mid t \ \mathbf{var} = \mathbb{E}; \ e \mid \mathbb{E}; \ e \\ &\mid \mathbf{announce}(v \dots \mathbb{E} e \dots); \ e \mid \mathbb{E} . \mathbf{register}() \end{aligned}$$

**Figure 22.** Added syntax, domains, and evaluation contexts used in the semantics, based on [10, 37].

$$\begin{aligned} (\mathbf{NCALL}) & \langle \langle \mathbb{E}[\mathbf{null}.m(v_1, \dots, v_n)], \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle \langle \mathbf{NullPointerException}, \tau \rangle + \psi, \mu, \gamma \rangle \\ (\mathbf{NGET}) & \langle \langle \mathbb{E}[\mathbf{null}.f], \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle \langle \mathbf{NullPointerException}, \tau \rangle + \psi, \mu, \gamma \rangle \\ (\mathbf{NSET}) & \langle \langle \mathbb{E}[\mathbf{null}.f = v], \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle \langle \mathbf{NullPointerException}, \tau \rangle + \psi, \mu', \gamma \rangle \\ (\mathbf{XCAST}) & \frac{[c'.F] = \mu(loc) \quad c' \not\prec: c}{\langle \langle \mathbf{cast} \ c \ loc \rangle, \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle \langle \mathbf{ClassCastException}, \tau \rangle + \psi, \mu, \gamma \rangle} \end{aligned}$$

**Figure 23.** Operational semantics of expressions that produce exceptions, base on, [10, 37].

from locations ( $loc$ ) to objects ( $o$ ). The subscriber list  $\gamma$  consists of a set of receiver objects for handler methods.

The task queue  $\psi$  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 local data ( $\tau$ ). The task local data is used to record the identity of the current task ( $n$ ) and a set of identities for other tasks on which this task depends on. A task  $t$  depends on another task  $t'$  if  $t$ 's read/write set conflicts with the read/write set of  $t'$ . Pāṇini will never schedule a task to run unless all the tasks in its dependence set (represented as the id set) 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 reference  $loc$ , which have standard meanings.

**Evaluation Contexts.** We present the semantics as a set of evaluation contexts  $\mathbb{E}$  (Figure 22) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context [48]. This avoids the need for writing out standard recursive rules and clearly presents the order of evaluation. The language uses the 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 24. These are mostly standard and adopted from the work of Rajan and Leavens [37], and Clifton’s work [10, 11].

$$\begin{array}{c}
\text{(NEW)} \\
\frac{loc \notin \text{dom}(\mu) \quad \mu' = \{loc \mapsto [c, \{f \mapsto \mathbf{null} \mid (t f) \in \text{fieldsOf}(c)\}]\} \oplus \mu}{\langle\mathbb{E}[\mathbf{new } c()]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[loc]\rangle, \tau + \psi, \mu', \gamma} \\
\text{(CALL)} \\
\frac{(c', t, m(t_1 \text{ var}_1, \dots, t_n \text{ var}_n)\{e\}, \rho) = \text{findMeth}(c, m) \quad [c.F] = \mu(loc) \quad e' = [\mathbf{this}/loc, \text{var}_1/v_1, \dots, \text{var}_n/v_n]e}{\langle\mathbb{E}[loc.m(v_1, \dots, v_n)]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[\mathbf{yield } e']\rangle, \tau + \psi, \mu, \gamma} \\
\text{(SEQUENCE)} \\
\langle\mathbb{E}[v; e]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[\mathbf{yield } e]\rangle, \tau + \psi, \mu, \gamma \\
\text{(CAST)} \quad \text{(DEFINE)} \\
\frac{[c'.F] = \mu(loc) \quad c' <: c}{\langle\mathbb{E}[\mathbf{cast } c \text{ loc}]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[loc]\rangle, \tau + \psi, \mu, \gamma} \quad \frac{e' = [var/v]e}{\langle\mathbb{E}[t \text{ var} = v; e]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[\mathbf{yield } e']\rangle, \tau + \psi, \mu, \gamma} \\
\text{(GET)} \quad \text{(SET)} \\
\frac{\mu(loc) = [c.F] \quad v = F(f)}{\langle\mathbb{E}[loc.f]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[v]\rangle, \tau + \psi, \mu, \gamma} \quad \frac{[c.F] = \mu(loc)}{\mu' = \mu \oplus (loc \mapsto [c.F \oplus (f \mapsto v)])} \\
\langle\mathbb{E}[loc.f = v]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle\mathbb{E}[v]\rangle, \tau + \psi, \mu', \gamma
\end{array}$$

**Figure 24.** Semantics of object-oriented expressions in Pāṇini, based in part on [10, 11, 37]

One difference stems from the concurrency and store models in Pāṇini. The use of the intermediate expression **yield** in the (CALL), (SEQUENCE), and (DEFINE) rules serves to allow other tasks to run. There are not any specific reasons for inserting the intermediate expression **yield** into the three above expressions and not the other, except that since thread interleaving is undeterministic and thus Pāṇini models this by undeterministically inserting the intermediate expression **yield** into different semantics rules.

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 (CALL) rule acquires the method signature using the auxiliary function *findMeth* (as *methodBody* in in Clifton’s work [10, 11]). 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 replaced by the actual values as well as the **this** variable by  $loc$ . It then yields

control by calling (YIELD) to model concurrency, which will be discussed later.

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 given to the value and evaluates the subsequent expressions with the new binding.

The (GET) or get field read rule gets an object record from the store and retrieves the corresponding field value as the result. The semantics for (SET) uses  $\oplus$  as an overriding operator for finite functions. That is, if  $\mu' = \mu \oplus (loc \mapsto v)$ , then  $\mu'(loc') = v$  if  $loc' = loc$  and otherwise  $\mu'(loc') = \mu(loc')$ . The operation first fetches the object from the store and overrides the field.

**Semantics for Yielding Control.** In Pāṇini’s semantics, like Abadi and Plotkin [2], the running task may implicitly relinquish control to other tasks. The rules for yielding control are given in Figure 25.

$$\begin{array}{c}
\text{(YIELD)} \\
\frac{\langle e', \tau' \rangle + \psi' = \text{active}(\psi + \langle \mathbb{E}[\mathbf{yield } e]\rangle, \tau)}{\langle \mathbb{E}[\mathbf{yield } e]\rangle, \tau + \psi, \mu, \gamma \hookrightarrow \langle \langle e', \tau' \rangle + \psi', \mu, \gamma \rangle} \\
\text{(YIELD-DONE)} \quad \text{(TASK-END)} \\
\frac{\langle \mathbb{E}[\mathbf{yield } e]\rangle, \tau + \bullet, \mu, \gamma}{\hookrightarrow \langle \mathbb{E}[e]\rangle, \tau + \bullet, \mu, \gamma} \quad \frac{\langle e', \tau' \rangle + \psi' = \text{active}(\psi) \quad \psi \neq \bullet}{\langle \langle v, \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle \langle e', \tau' \rangle + \psi', \mu, \gamma \rangle}
\end{array}$$

**Figure 25.** Semantics of yielding control in Pāṇini

The (YIELD) rule says to put the current task configuration in the end of the task-queue and to start evaluating the next active task configuration from the current task queue. Finding an active task is done by the auxiliary function *active* (shown in Figure 26). It searches the items (task configuration) from the head of the queue until it finds the first item that could be run and returns it. A task configuration is ready to run if the tasks in its dependence set are done.

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

**Figure 26.** Auxiliary functions for returning a nonblock configuration.

The (YIELD-DONE) rule is applied when there are no other task configurations in the queue. Since there are no other tasks in the queue, it continues to evaluate the current task configuration. The (YIELD-END) rule says that the current running task is done (it evaluates to a single value  $v$ ), thus this task configuration is removed from the queue and next active task will be scheduled.

**Semantics for Event registration.** We now describe the semantics for subscribing to an event (Figure 27).

$$\begin{array}{c}
\text{(MULTI-REGISTER)} \qquad \text{(REGISTER)} \\
\frac{loc \in \gamma}{\langle \langle \mathbb{E}[\mathbf{register}(loc)], \tau \rangle + \psi, \mu, \gamma \rangle} \quad \frac{loc \notin \gamma}{\langle \langle \mathbb{E}[\mathbf{register}(loc)], \tau \rangle + \psi, \mu, \gamma \rangle} \\
\hookrightarrow \langle \langle \mathbb{E}[loc], \tau \rangle + \psi, \mu, \gamma \rangle \quad \hookrightarrow \langle \langle \mathbb{E}[loc], \tau \rangle + \psi, \mu, loc + \gamma \rangle \\
\\
\text{(ANNOUNCE)} \\
\frac{\mathbf{event } p\{t_1 \text{ var}_1, \dots, t_n \text{ var}_n\} = CT(p) \quad \psi' = \psi + \psi' \quad \tau = \langle id, I' \rangle \quad \tau' = \langle id, I \rangle}{\nu = v_1 + \dots + v_n \quad \langle \psi'', I \rangle = spawn(p, \psi, \gamma, \nu, \mu)} \\
\frac{\langle \langle \mathbb{E}[\mathbf{announce } p(v_1, \dots, v_n); e], \tau \rangle + \psi, \mu, \gamma \rangle}{\hookrightarrow \langle \langle \mathbb{E}[\mathbf{yield } e], \tau' \rangle + \psi', \mu, \gamma \rangle}
\end{array}$$

**Figure 27.** Semantics of Registration and Announcement

The (MULTI-REGISTER) rule is applied when the handler has already registered previously and thus the configuration does not change. Pāṇini does not allow multiple registrations for a same object because in many circumstances different handlers of the same class may change a same field of their own handler objects and thus no conflict between them, unless the handler objects are the same.

The (REGISTER) rule finds out that this handler is not in the queue, so Pāṇini safely puts this record at the front of the queue.

**Semantics for announcing an event.** The semantics for signaling events is shown in Figure 27.

The (ANNOUNCE) rule takes the relevant event declaration from the program's list of declarations and creates a list of actual parameters ( $\nu$ ). This list of actual parameters ( $\nu$ ) is used by the auxiliary function  $spawn$  shown in Figure 28 (with other helper functions in Figure 29 and Figure 30). This rule does not change the ordering in existing task configurations

The auxiliary function  $concat$  is used in several other auxiliary functions. It (defined in Figure 30) combines the contents in the two lists, which are the inputs to this function.

The function  $spawn$  searches the program's global list of subscribers ( $\gamma$ ) for applicable handlers (using auxiliary functions  $hfind$ ,  $hmatch$ , and  $match$ ). Auxiliary functions  $buildconfs$  (Figure 29) and  $buildconf$  create task configurations for handlers.  $buildconf$  binds the context variables (of the event type) with the values ( $\nu$ ), computes an unique id for each handler task, and configures the dependent set of this handler (discussed in 2.3). These task configurations are used to run the handler bodies and they are appended to the

$$\begin{array}{l}
spawn(p, \psi, \gamma, \nu, \mu) = buildconfs(H, \psi, \nu, \bullet, \gamma, \mu) \\
\text{where } H = hfind(\gamma, p, \mu) \text{ and } CT \text{ is the program's list of declarations}
\end{array}$$

$$\begin{array}{l}
hfind(\bullet, p, \mu) = \bullet \\
hfind(loc + \gamma, p, \mu) = hfind(\gamma, p, \mu) \\
\text{where } \mu(loc) = [c.F] \text{ and } hmatch(c, p, CT) = \bullet \\
hfind(loc + \gamma, p, \mu) = concat(hfind(\gamma, p, \mu), \langle loc, m \rangle) \\
\text{where } \mu(loc) = [c.F] \text{ and } hmatch(c, p, CT) = m \\
\text{and } CT \text{ is the program's list of declarations}
\end{array}$$

$$\begin{array}{l}
hmatch(c, p, \bullet) = \bullet \\
hmatch(c, p, (\mathbf{event } p\{\dots\}) + CT') = hmatch(c, p, CT') \\
hmatch(c, p, (\mathbf{class } c' \dots) + CT') = hmatch(c, p, CT') \text{ where } c \neq c' \\
hmatch(c, p, ((\mathbf{class } c \text{ extends } d \dots \text{ binding}_1 \dots \text{ binding}_n) + CT')) \\
= excl(match((binding_n + \dots + binding_1), p), hmatch(d, p, CT)) \\
\text{where } excl(\bullet, H) = H \text{ and } excl(e, H) = e
\end{array}$$

$$\begin{array}{l}
match(\bullet, p) = \bullet \\
match((\mathbf{when } p' \text{ do } m) + B, p) = match(H, p) \text{ where } p' \neq p \\
match((\mathbf{when } p' \text{ do } m) + B, p) = m
\end{array}$$

**Figure 28.** Functions for Creating Task Configurations.

$$\begin{array}{l}
buildconfs(\bullet, \psi, \nu, H', \gamma, \mu) = (\bullet, \bullet) \\
buildconfs(\langle loc, m \rangle + H, \psi, \nu, H', \gamma, \mu) \\
= (\langle e, \langle m_{id}, I \rangle \rangle + \psi', concat(m_{id}, I')) \\
\text{where } \langle e, \langle m_{id}, I \rangle \rangle = buildconf(loc, m, \psi, \nu, H', \gamma, \mu) \\
\text{and } H'' = H' + \langle loc, m \rangle \\
\text{and } (\psi', I') = buildconfs(H, \psi, \nu, H'', \gamma, \mu)
\end{array}$$

$$\begin{array}{l}
buildconf(loc, m, \psi, \nu, H, \gamma, \mu) = \\
\text{let } e' = [\mathbf{this}/loc, var_1/v_1, \dots, var_n/v_n]e \text{ in } \langle e', \langle id, I \rangle \rangle \\
\text{where } loc = [c.F] \text{ and} \\
(c', t, m(t_1 \text{ var}_1, \dots, t_n \text{ var}_n)\{e\}, \dots) = findMeth(c, m) \\
\text{and } \nu = v_1 + \dots + v_n \text{ and } I = pre(loc, m, H, id' + 1, \gamma, \mu) \\
\text{and } id = 1 + car(H) + id' \text{ and } id' = max(\psi, -1)
\end{array}$$

$$\begin{array}{l}
pre(loc, m, \bullet, n, \gamma, \mu) = \bullet \\
pre(loc, m, \langle loc_1, m_1 \rangle + H, n, \gamma, \mu) = pre(loc, m, H, n + 1, \gamma, \mu) \\
\text{where } loc = [c.F] \text{ and } (c', t, m \dots, \rho) = findMeth(c, m) \\
\text{and } loc_1 = [c_1.F] \text{ and } (c'_1, t_1, m_1 \dots, \rho') = findMeth(c_1, m_1) \\
\text{and } true = diff(update(\rho, \gamma, \mu), update(\rho', \gamma, \mu)) \\
pre(loc, m, \langle loc_1, m_1 \rangle + H, n) = concat(n, pre(loc, m, H, n + 1)) \\
\text{where } loc = [c.F] \text{ and } (c', t, m \dots, \rho) = findMeth(c, m) \\
\text{and } loc_1 = [c_1.F] \text{ and } (c'_1, t_1, m_1 \dots, \rho') = findMeth(c_1, m_1) \\
\text{and } false = diff(update(\rho, \gamma, \mu), update(\rho', \gamma, \mu))
\end{array}$$

$$\begin{array}{l}
update(\bullet, \gamma, \mu) = \bullet \\
update(\mathbf{read } cf) + \rho, \gamma, \mu = concat(\langle \mathbf{read } cf \rangle, update(\rho, \gamma, \mu)) \\
update(\mathbf{write } cf) + \rho, \gamma, \mu = concat(\langle \mathbf{write } cf \rangle, update(\rho, \gamma, \mu)) \\
update(\mathbf{create}) + \rho, \gamma, \mu = concat(\langle \mathbf{create} \rangle, update(\rho, \gamma, \mu)) \\
update(\mathbf{reg}) + \rho, \gamma, \mu = concat(\langle \mathbf{reg} \rangle, update(\rho, \gamma, \mu)) \\
update(\mathbf{ann}) + \rho, \gamma, \mu = concat(\langle \mathbf{ann} \rangle, update(\rho, \gamma, \mu)) \\
\text{getE}(hfind(\gamma, p, \mu), \gamma, \mu, update(\rho, \gamma, \mu))
\end{array}$$

**Figure 29.** Functions for building handler configurations.

end of the queue  $\psi$ . The auxiliary function  $max$  is used to give the newly-born task a global unique ID.

The auxiliary function  $pre$  is used to find all the tasks that this task depends on. It first calls another auxiliary function  $update$  to update the effects of the task. The function  $update$  is used because the handler may signal events, say  $e$ , thus this function searches the handler queue  $\gamma$  to union their effect sets with the effect set of this task. Pāṇini does this to get more accurate information about the potential effect sets of a task to reduce false conflicts. The functions  $diff$  and

$$\begin{aligned}
& \text{car}(\bullet) = 0 \\
& \text{car}(\langle \text{loc}, m \rangle + H) = 1 + \text{car}(H) \\
& \text{max}(\bullet, id) = id \\
& \text{max}(\langle e', \langle id', I \rangle \rangle + \psi, id) = \text{max}(\psi, id) \text{ where } id' < id \\
& \text{max}(\langle e', \langle id', I \rangle \rangle + \psi, id) = \text{max}(\psi, id') \text{ where } id' > id \\
& \text{getE}(\bullet, \gamma, \mu) = \bullet \\
& \text{getE}(\langle \text{loc}, m \rangle + H, \gamma, \mu) = \text{concat}(\text{update}(\rho, \gamma, \mu), \text{getE}(H, \gamma, \mu)) \\
& \quad \text{where } \text{loc} = [c.F] \text{ and } (c', t, m \dots, (\dots, \rho) \text{ in } c') = \text{findMeth}(c, m) \\
& \text{diff}(\bullet, \rho) = \text{true} \\
& \text{diff}(\epsilon + \rho', \rho) = \text{diff}(\rho', \rho) \text{ where } \text{true} = \text{differ}(\epsilon, \rho) \\
& \text{diff}(\epsilon + \rho', \rho) = \text{false} \text{ where } \text{false} = \text{differ}(\epsilon, \rho) \\
& \text{differ}(\epsilon, \bullet) = \text{true} \\
& \text{differ}(\epsilon, \epsilon' + \rho) = \text{differ}(\epsilon, \rho) \text{ where } \epsilon \text{ and } \epsilon' \text{ have no conflict} \\
& \text{differ}(\epsilon, \epsilon' + \rho) = \text{false} \text{ where } \epsilon \text{ and } \epsilon' \text{ have conflicts} \\
& \text{concat}(\bullet, L') = L' \\
& \text{concat}(l + L, L') = l + \text{concat}(L, L')
\end{aligned}$$

**Figure 30.** Miscellaneous helper functions.

*diff* are used to actually compare the effects. The table in Figure 17 is used to compare effects. A read effect will conflict with a write effect if they access the same field of the same class or a subclass of another class. A read effect also conflicts with the register effect. A write effect will conflict with another write effect similar to the read effect discuss above. An announce effect conflicts with only register effects and register effects will conflict with any effect except for the create effect.

## C. Properties of Pāṇini’s Design

**Definition C.1.** [Blocked Configurations.] *A task configuration  $\langle \langle e, \tau \rangle + \psi, \mu, \gamma \rangle$  may block if any one of its predecessors<sup>5</sup> is still in execution.*

**Theorem C.2.** [Liveness.] *Let  $\langle \langle e, \tau \rangle + \psi, \mu, \gamma \rangle$  be an arbitrary program configuration, where  $e$  is a well-typed expression,  $\tau$  is task local data,  $\mu$  is the store,  $\psi$  is a task queue and  $\gamma$  is a handler queue. Then either  $e$  is not blocked or there is some task configuration in  $\psi$  that is not blocked.*

*Proof Sketch:* We could construct a tree using the tasks, where any parent node,  $p$ , publishes an event,  $E$ , and the handlers of  $E$  form the children of  $p$ . So, in this case, nodes in a lower level will never depend on nodes in the above levels. A node may depend on its children when it is publishing an event or it may depend on a sibling if its effect set conflicts with the sibling’s. On any particular level of the tree, if siblings conflict with each other, then the initial registration order (Section 2.3) is used to create a non-blocking ordering for the handlers. Finally, leaf nodes, which have no child de-

<sup>5</sup> a task  $t_1$  is a predecessor of another task  $t_2$ , if either 1)  $t_2$  depends on  $t_1$ , which means that the effect set of  $t_2$  conflicts with the effect set of  $t_1$ , as mentioned in subsection 2.3, or 2) if  $t_2$  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 discribed in Section B).

pendencies, can always either be run concurrently or in an ordering determined by registerstration time/sibling dependency. 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 type Pāṇini program does not deadlock.

### C.1 Proof of Type Soundness

A standard preservation and progress argument [48] is used to prove the soundness of Pāṇini’s type system. The details are adapted from previous work [10, 11, 19]. 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  $\rho$  in subsequent discussion. The key definition of consistency is as follows.

**Definition C.3.** [Environment-Store Consistency.] *Suppose we have a type environment  $\Pi$  and  $\mu$  a store. Then  $\mu$  is consistent with  $\Pi$ , written  $\mu \approx \Pi$ , if and only if all the followings hold:*

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

**LEMMA C.4.** [Substitution.] *If  $\Pi, \text{var}_1 : t_1, \dots, \text{var}_n : t_n \vdash e : t$  and  $\forall i \in \{1..n\} \cdot \Pi \vdash e_i : s_i$  where  $s_i <: t_i$  then  $\Pi \vdash [\text{var}_1/e_1, \dots, \text{var}_n/e_n]e : s$  for some  $s <: t$ .*

*Proof Sketch:* Let  $\Pi' = \Pi, \text{var}_1 : t_1, \dots, \text{var}_n : t_n$  and  $[\text{var}'/e'] = [\text{var}_1/e_1, \dots, \text{var}_n/e_n]$ . 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-NUL) and (T-VAR), which have no variables and  $s = t$ . Other cases can be proved by adaptations of MiniMAO<sub>0</sub> [10]. 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-SET-LOCAL), (T-CALL), (T-GET) and (T-GET-LOCAL) are similar to Clifton’s proofs. We now consider the case for (T-REGISTER), (T-ANNOUNCE) and (T-YIELD).

For **announce**  $p(e_1, \dots, e_n); e_{n+1}$ , 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. Also, by IH, the substitution of  $e_{n+1}, [\text{var}'/e']e_{n+1}$  has the subtype of  $e_{n+1}$ . Therefore, since the whole expression has the same type as  $e_{n+1}$ , 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  $e$ . ■

We now state standard lemmas for environment contraction, replacement and replacement with subtyping. These lemmas can be proved by adaptations of Clifton’s proofs for MiniMAO<sub>0</sub> [10]. We omit them here.

LEMMA C.5. [Environment Extension.] *If  $\Pi \vdash e : t$  and  $a \notin \text{dom}(\Pi)$ , then  $\Pi, a : t' \vdash e : t$ .*

LEMMA C.6. [Environment Contraction.] *If  $\Pi, a : t' \vdash e : t$  and  $a$  is not free in  $e$ , then  $\Pi \vdash e : t$ .*

LEMMA C.7. [Replacement.] *If  $\Pi \vdash \mathbb{E}[e] : t, \Pi \vdash e : t'$ , and  $\Pi \vdash e' : t'$ , then  $\Pi \vdash \mathbb{E}[e'] : t$ .*

LEMMA C.8. [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  $t' <: t$ .*

**Theorem C.9.** [Progress.] *For a well-typed expression  $e$ , a task local data  $\tau$ , a task queue  $\psi$ , a store  $\mu$ , and a handler queue  $\gamma$ . If  $\Pi \vdash e : t$  and  $\mu \approx \Pi$ , then either*

- $e = \text{loc}$  or
- $e = \mathbf{null}$  or  $e = \text{NullPointerException}$  or  $e = \text{ClassCastException}$  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  $e = \mathbf{null}$ , it is trivial.

(b) Cases  $e = \text{NullPointerException}$  or  $e = \text{ClassCastException}$  result from the semantics rules  $\mathbf{null}.f$ ,  $\mathbf{null}.f = v$ ,  $\mathbf{null}.m(v_1, \dots, v_n)$ ,  $\mathbf{register}(\mathbf{null})$  and  $\text{cast } e$  (shown in Figure 23). These values serve as the base cases.

(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  $e = \mathbb{E}[\mathbf{new } c()]$ ,  $e = \mathbb{E}[\text{loc}.m(v_1, \dots, v_n)]$ ,  $e = \mathbb{E}[\text{loc}.f]$ ,  $e = \mathbb{E}[\text{loc}.f = v]$ ,  $e = \mathbb{E}[\mathbf{cast } t \text{ loc}]$ ,  $e = \mathbb{E}[t \text{ var} = v; e]$  and  $e = \mathbb{E}[v; e_1]$  are similar to Clifton’s work [10, 11] and are omitted.

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

Case  $e = \mathbb{E}[\mathbf{announce } p(v_1, \dots, v_n); \{e\}]$ . 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 C.10.** [Subject-reduction.] *Let  $e$  be an expression and  $e \neq \mathbf{yield } e_1$  for any  $e_1$ ,  $\tau$  task local,  $\psi$  a task queue,  $\mu$  a store, and  $\gamma$  a handler queue. Let  $\mu \approx \Pi$  be a type environment and  $t$  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) and (CALL)) can be proved by adaptations of Clifton’s proofs for MiniMAO<sub>0</sub> [10, Section 3.1.4].

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),  $e = \mathbb{E}[t \text{ var} = v; e_1]$  and  $e' = \mathbb{E}[[\text{var}/v]e_1]$ : 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  $t \text{ var} = v; e_1$  and all its subterms are well typed in  $\Pi$ . Let  $\Pi \vdash (t \text{ var} = v; e_1) : u$ . By (T-Define),  $\Pi, \text{var} : t \vdash e_1 : u'$ . By Lemma C.4,  $\Pi \vdash [\text{var}/v]e_1 : u''$  for some  $u'' <: u' <: u$ . Therefore, by lemma C.8,  $\Pi \vdash e' : t'$  for some  $t' <: t$ .

For the (MULTI-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  $e = \mathbb{E}[\mathbf{announce } p\{v_1, \dots, v_n\}; e_2]$ . Let  $\mu' = \mu$ ,  $\gamma' = \gamma$ ,  $\Pi' = \Pi$  and  $t' = t$ . Thus  $\Pi \vdash e_2 : t$ , has the same type as  $\Pi \vdash \mathbf{yield } e_2 : t$ . ■

**Definition C.11.** [Thread-interleaving.] *If*

$$\begin{aligned} &\langle\langle \mathbb{E}[\mathbf{yield } e], \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle\langle \mathbb{E}_1[e_1], \tau_1 \rangle + \psi_1, \mu_1, \gamma_1 \rangle \\ &\dots \\ &\hookrightarrow \langle\langle \mathbb{E}_n[e_n], \tau_n \rangle + \psi_n, \mu_n, \gamma_n \rangle \hookrightarrow \langle\langle \mathbb{E}[e], \tau \rangle + \psi', \mu', \gamma' \rangle \\ &\text{or} \\ &\langle\langle \mathbb{E}[\mathbf{yield } e], \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow \langle\langle \mathbb{E}[e], \tau \rangle + \psi', \mu', \gamma' \rangle, \\ &\text{we denote this as } \langle\langle \mathbb{E}[\mathbf{yield } e], \tau \rangle + \psi, \mu, \gamma \rangle \hookrightarrow^* \\ &\langle\langle \mathbb{E}[e], \tau \rangle + \psi', \mu', \gamma' \rangle, \text{ where } \forall i \{1 \leq i \leq n\} \mathbb{E}_i[e_i] \neq \mathbb{E}[e]. \end{aligned}$$

**Theorem C.12.** [Subject-reduction-Thread-interleaving.] *For an expression  $e = \mathbb{E}[\mathbf{yield } e_1]$  for any  $e_1$ ,  $\tau$  task local data, and  $\psi$  a task queue,  $\mu$  a store and  $\gamma$  a handler queue. Let  $\mu \approx \Pi$  be a type environment and  $t$  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  $\mu' \approx \Pi$  and  $\Pi \vdash e' : t'$  and  $t' <: t$ .*

*Proof Sketch:* The proof is based on the observation that Pāñini does not have data race (because when an event is announced, Pāñini schedules tasks to eliminate races and to maximize concurrency, as discussed in 2.3) and thus, 1 of Definition C.3 holds. Since the store  $\mu$  does not shrink, 2 of Definition C.3 holds. Clearly,  $\mathbf{yield } e_1$  and  $e_1$  have the same type in  $\Pi$ , and therefore, by Lemma C.7, if  $\Pi \vdash e : t$ , then  $\Pi \vdash e' : t'$ . ■

**Theorem C.13.** [Soundness.] *Given a program*

$P = \mathbf{decl } 1 \dots \mathbf{decl } n \text{ e}$ , *if  $\vdash P : (t, \rho)$  for some  $t$  and  $\rho$ , then either the evaluation of  $e$  diverges or else  $\langle\langle e, \langle 0, \emptyset \rangle \rangle + \bullet, \bullet \rangle \hookrightarrow^* \langle\langle e', \tau' \rangle + \psi', \mu', \gamma' \rangle$  where one of the*

*following holds for v:*

- $e = loc$  or
- $e = \mathbf{null}$  or
- $e = NullPointerException$  or
- $e = ClassCastException$ .

*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 C.9, Theorem C.10 and Theorem C.12. ■