Implicit invocation meets safe, implicit concurrency

by

Yuheng Long

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. I would like to thank Hridesh Rajan for his guidance, patience and support throughout this research and the writing of this thesis. Thanks are due to the US National Science Foundation for financially supporting this project under grant CCF-08-46059.

I would like to thank my committee members Gurpur M. Prabhu and Steve Kautz for their efforts and contributions to this work. Also, I would like to thank the reviewers of GPCE 2010 conference for their insightful feedback. I would like to extend my thanks to all the members of Laboratory of Software Design for offering constructive criticism and timely suggestions during research. Additionally, I would like to thank Sean L. Mooney for his help on implementing the compiler and Tyler Sondag for writing examples and analyzing experimental data.

I am very grateful to my parents for their moral support and encouragement throughout the duration of my studies.

CHAPTER 1. Introduction

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 Explicit 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 [6, 44, 46, 62]. 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 [32]. Several such language features exist in common language libraries. For example, threads, `Future`s, and `FutureTask`s are all included in the Java programming language's standard library [46, 62]. Using such libraries has advantages, e.g. they can encapsulate complex synchronization code and allow its reuse. However, their main disadvantage is that today they do not provide guarantees such as race freedom, deadlock freedom and sequential semantics. This makes it much harder and error prone to write correct concurrent programs.

To illustrate, consider the implementation of a genetic algorithm in Java presented in Figure 1.1. 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 [55]. The main concept is that searching for a desirable state is done by combining two *parent* states instead of modifying a single state [55]. An initial *generation* with n members is given to the algorithm. Next, a *crossover* function is used to combine different members of the generation in order to develop the next generation (lines 10–16 in Figure 1.1). Optionally, members of the offspring may randomly be *mutated* slightly (lines 18–23 in Figure 1.1). Finally, members of the generation (or an entire generation) are ranked using a *fitness function* (lines 25–29 in Figure 1.1).

Multiple Concerns of the Genetic Algorithm. In the OO implementation of the genetic algorithm in Figure 1.1, there are three concerns standard to the genetic algorithm: crossover (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 in the example is concurrency (lines 4, 7–9, and 30–33). In this example, production of a generation is run as a `FutureTask`, but other solutions are also possible. The shading represents different concerns as illustrated in the legend.

1.2 Problems with Explicit Concurrency Features

Explicit concurrency. With explicit concurrency, programmers must divide the program into independent tasks. Next, they must handle creating and managing threads. A problem with the concurrency-unsafe language features described above and illustrated in Figure 1.1 is that correctness is difficult to ensure since it relies on all objects obeying a usage policy [33]. Since such policies cannot automatically be enforced by a library based approach [33], the burden on the programmers is increased and errors arise (e.g., deadlock, races, etc.). Also, the non-determinism introduced by such mechanisms makes debugging hard since errors are difficult to reproduce [59]. Furthermore, this style of explicit parallelism can hurt the design and maintainability of the resulting code [53].

Separation of modular and concurrent design. Another 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 1.1, 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

Legend Concurrency Logging Mutation Crossover 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     executor.execute(t);
31     try { return t.get(); }
32     catch (InterruptedException e) { return  g; }
33     catch (ExecutionException e) { return g; }
34   }
35 }
```

Figure 1.1 Genetic algorithm with Java concurrency utilities

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 (crossover, mutation, logging, and fitness calculation).

1.3 Contributions

Our language, Pāñini, addresses these problems. The key idea behind Pāñini's design is to provide programmers with mechanisms to utilize prevalent idioms in modular program

```
event GenAvailable {
  Generation g; //Reflective information available at events
}
class CrossOver {
  int probability; int max;
  CrossOver(...){
    register(this);
    // initialization elided (initializes fields above).
  }
  when GenAvailable do cross;
  void cross(Generation g) {
    int gSize = g.size();
    Generation g1 = new Generation(g);
    for(int i = 0; i< gSize; i+=2){
      Parents p = g.pickParents();
      g1.add(p.tryCrossOver(probability));
    }
    if(g1.getDepth() < max) announce GenAvailable(g1);
}}
class Mutation {
  int probability; int max;
  Mutation(...){
    register(this);
    // initialization elided (initializes fields above).
  }
  when GenAvailable do mutate;
  void mutate(Generation g) {
    int gSize = g.size();
    Generation g2 = new Generation(g);
    for(int i = 0; i< gSize; i+=2){
      Parents p = g.pickParents();
      g2.add(p.tryMutation(probability));
    }
    if(g2.getDepth() < max) announce GenAvailable(g2);
}}
class Logger {
  when GenAvailable do logit;
  Logger(){ register(this); }
  void logit(Generation g) { logGeneration(g); }
}
class Fittest {
  Generation last;
  when GenAvailable do check;
  Fittest(){ register(this); }
  void check(Generation g) {
    if(last == null) last = g;
    else {
          Fitness f1 = g.getFitness();
          Fitness f2 = last.getFitness();
          if(f1.average() > f2.average()) last = g;
    }
  }
}
```

Figure 1.2 Pāñini's version of the Genetic algorithm

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 Pāñini. An *asynchronous, typed event* exposes potential concurrency in programs which use behavioral design patterns for object-oriented languages, e.g., the observer pattern [24]. These patterns are widely adopted in software systems such as graphical user interface frameworks, middleware, databases, and Internet-scale distribution frameworks.

In Pāñini, 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 describe the syntax in the context of the genetic algorithm implementation in Pāñini shown in Figure 1.2 (a more detailed description appears in Chapter 3). In the listing we have omitted initializations for brevity. In this listing an example of an event type appears on lines 1–3, whose name is `GenAvailable` and that declares one *context variable* `g` of type `Generation` on line 2. Context variables define the reflective information available at events of that type.

Certain classes, 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 crossover 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 1.2 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 Pāñini classes statically express (potential) interest in an event by providing a *binding declaration*. For example, the `Mutate` concern (lines 20-35) wants to randomly change 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* statements. The class `Mutation` has a constructor 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) is run with the registered instance `this` as the receiver object.

Concurrently, the method `logit` (line 39) in class `Logger` will log each generation and the method `check` in class `Fittest` (lines 41-51) will determine the better fitness between the announced generation and the previously optimal generation.

Benefits of Pāñini's Implementation. 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 Pāñini's implementation encapsulates concurrency management code. By not requiring users to write this code, Pāñini avoids any threat of incorrect or non-deterministic concurrency, thus easing the burden on programmers. This allows them to focus on creating a good, maintainable modular design.

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

Advantages of Pāñini's Design over Related Ideas. Pāñini is most similar to our previous work on Ptolemy [50], but Pāñini's event types also have concurrency advantages. Compared to similar ideas for aspect-oriented advice presented by Ansaloni *et al.* [4], Pāñini only exposes concurrency safely.

It is also similar to implicit invocation (II) languages [16, 42] that also see *events* as a decoupling mechanism. The advantage of using Pāñini over an II language is that asynchronous, typed events in Pāñini allow developers to take advantage of the decoupling of subjects and observers to expose potential concurrency between their executions. A detailed comparison is presented in Chapter 2.

Pāñini also relieves programmers from the burden of explicitly creating and maintaining threads, managing locks and shared memory. Thus it 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 Pāñini's runtime, thus thrashing due to excessive threading is avoided. These benefits make Pāñini an interesting point in the design space of concurrent languages.

In summary, this work makes the following contributions:

1. Pāñini's language design that reconciles implicit-invocation design style and implicit concurrency and provides a simple and flexible concurrency model such that Pāñini programs are
    - free of data races,
    - free of deadlocks, and
    - have a guaranteed deterministic semantics(a given input is always expected to produce the same output.[47]);
2. an efficient implementation of Pāñini's design as an extension of the JastAdd compiler[18] that relies on:
    - an 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 [34];
3. a detailed analysis of Pāñini and closely related ideas;
4. and, an empirical performance analysis using canonical concurrency examples implemented using Pāñini and using standard techniques which shows that the performance and scalability of the implementations are comparable.

1.4 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2: This chapter surveys related work of observer pattern, implicit concurrency and explicit concurrency. **Chapter 3**: This chapter gives the formal definition of Pāñini's syntax and the design. **Chapter 4**: This chapter provides the type system for Pāñini. **Chapter 5**: The detail operational semantics are presented. **Chapter 6**: Proofs of Pāñini's soundness is given that any valid Pāñini program is free of data races and dead lock. At the same time, it provides programmers with deterministic semantics. **Chapter 7**: We will describe Pāñini's compiler and runtime system. JastAdd compiler was extended to compile Pāñini's programs. Also, we will explain how Pāñini programs make use of the underlying fork/join framework [34]. **Chapter 8**: This chapter describes our performance evaluation and experimental results. As is shown, the implementation of this language has low overhead and exposes useful concurrency as expected. Also, more examples in Pāñini are presented. It offers some ideas as the expressiveness of the proposed language. **Chapter 9**: This chapter concludes the thesis. Also, it will briefly describes future directions.

CHAPTER 2. Related Work

In this chapter, we discuss previous and current work in the field of implicit invocation and concurrent programming.

2.1 Implicit Invocation Languages

Events have a long history in both the software design [58, 42, 22, 37, 15] and distributed systems communities [21]. Pāñini's notion of asynchronous, typed events build on these notions, in particular recent work in programming languages focusing on event-driven design [19, 20, 50]. In software design, events and implicit-invocation have been seen as a decoupling mechanism for modules [58, 42], whereas in distributed systems, events are seen as a mechanism of decoupling component execution for location transparent deployment and extensibility [43, 56].

A key difference between the programming models developed for event-based systems/message-passing systems/actor-based languages and that of Pāñini is that the former assume that components in the system do not share state and only communicate by passing value types or record of value types [43, 21, 31, 5], whereas the latter allows shared states (similar to mainstream languages like Java, C#) that is useful for many computation patterns. This means that if features from the former are adopted to mainstream languages as it is to decouple execution of components participating in an implicit-invocation design style, programmers will be directly responsible for ensuring that concurrent components do not have data races and deadlocks. Furthermore, reasoning about such systems will also be difficult due to concurrency [6, 44]. In Pāñini, programmers get concurrency benefits as a direct result of good design. Previous work on message-passing, publish/subscribe and actor-based languages either require programmers to manually account for data races, or have a sequential model or assume disjoint address space between concurrent processes [56, 5].

2.2 Implicitly Concurrent Languages

Like Jade [53], Pāñini is an implicit concurrency language. Programmers in Jade supply information about the effect of tasks so that the compiler may discover concurrency. Pāñini is different in that it automates the process and removes the burden on the programmer to supply these effects by hand. Pāñini also removes any errors which could be introduced by incorrect specification of effects. Pāñini detects conflict when handlers register.

In POOL [3], ABCL [64], Concurrent Smalltalk [65] and BETA [57], objects implicitly execute in the context of a local process. This is different from Pāñini 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 [37]. In this work, we do not discuss the semantic issues with this integration, however.

2.3 Explicitly Concurrent Languages

Pāñini is different from Grace [7] which is an explicit threading language. Grace executes threads speculatively. If a conflict is detected, it rolls back the changes. Otherwise it commits the changes. Like X10 [45], Pāñini 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) [35], wraps computation into tasks and uses thread stealing as the underlying implementation. This is similar to Pāñini's runtime, but programmers in TPL have to explicitly account for races, whereas Pāñini automatically avoids all races.

Similar to the effect sets of Pāñini, deterministic parallel Java (DPJ/DPJizer) [47, 41] uses effect sets to provide deterministic semantics for programs. For DPJ/DPJizer, programmers explicitly write annotations on object fields, which ensure that fields are in separate regions.

Then the tool infers summary for methods. Pāñini does not require any specification. DPJ provides programmers with two concurrent constructs to parallelize their programs. This is unlike Pāñini, which does not require programmers to construct explicitly parallel programs. Instead, Pāñini promotes the goal of writing programs with good modular designs.

Unlike Multilisp [54], which has the future construct, Pāñini uses different expressions as synchronization points. Moreover, unlike Java's current adoption of Futures, which is unsafe [62], heap access expressions in Pāñini are sound. Furthermore, unlike previous work [46, 62], our implementation doesn't require modifications to the virtual machine.

Other recent work such as TaskJava [22] and Tame [31], have promoted similar integration with existing languages. For TaskJava, an `asynchronous` method is marked with `async`, indicating that it could block. 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. Similarly, Tame uses a primitive `twait` to block on events. In both these approaches, running of the concurrent task is explicitly managed by the programmer. In Pāñini, 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.

2.4 Types, Regions and Effects

FX [28, 38] is a Scheme-like, implicitly parallel language. Unlike Pāñini, it used region-based types and effects for concurrency. Effects for object-oriented language was first studied by Greenhouse and Boyland [27]. However, they did not apply their work to concurrency. Also, unlike Greenhouse and Boyland's work Pāñini does not rely entirly on objects located in different regions.

Ownership system has been studied for more than ten years [11]. Clarke and Drossopoulou in their work [12] described an effect system and use it to reason about the absence of aliasing. Multiple Ownership for Java-like Objects (MOJO) [11] introduced the notion of multiple ownership and enforced the "objects in boxed" model. All of these works require that objects have some relationship with others, i.e. certain objects belong solely to some other objects. Because of this, region creation is couple with object creation. Unlike any of the above works, Pāñini does not require programmers to write any effect annotation/type or ownership relation on the programs. Rather, effects summaries of methods are automatically inferred.

Effects systems [1, 9, 30] are also used in explicit concurrent languages to eliminate data races and deadlocks. Unlike any of the above systems, which made use of the effects system to enforce certain locking discipline, Pāñini is an implicit concurrent language and has no syntax for locks. What is more, Pāñini provides a deterministic semantics.

Many explicit concurrent languages benefit from using sophisticated type systems. AJ [61] lets programmers to specify that a set of fields must be accessed atomically, but does not guarantee a deterministic semantics. Concurrent Revisions [10] provided users with a syntax that says each thread accesses its own version of certain objects to eliminate interferences. However, the underlying implementation only does a shallow copy, which means that programmers have to explicitly put every object that could produce interferences into the revisions. Pāñini tries to relieve programmers from concurrency bugs and is also an implicit concurrent language.

CHAPTER 3. Pāñini's Design

In this chapter, we describe Pāñini's design. Pāñini's design builds on our previous work on the Ptolemy [50] and Eos [52] languages as well as implicitly parallel languages such as Jade [53]. Pāñini achieves concurrent speedup by executing handler methods concurrently. The novel features of Pāñini are found in its concurrency model and conflict-detection scheme.

3.1 Pāñini's Syntax

Pāñini extends Java [26] with new mechanisms for declaring events and for announcing these events. These features are inspired by implicit invocation (II) languages such as Rapide [16] and our previous work on Ptolemy [50]. These syntax extensions are shown in Figure 3.1.

In this syntax, the novel features are: event type declarations (`event`), event announcement statements (`announce`), and handler registration statements (`register`). Since Pāñini is an implicitly concurrent language, it does not feature any construct for spawning threads or for mutually exclusive access to shared memory. Rather, concurrent execution is facilitated by announcing events, using the `announce` statement, which may cause handlers to run concurrently. Examples of the syntax can be seen in Figure 1.2 and described in Section 1.3.

Top-level Declarations. The object-oriented part of Pāñini has classes, objects, inheritance, and subtyping, but it does not have `super`, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods Figure 3.1. A Pāñini program consists of a sequence of declarations followed by an expression, which can be thought of as the body of a "main" method. We add a new declaration for events. An event type declaration (*EventDecl*) has a name (*Identifier*), and zero or more context variable declarations (*ContextVariable**).

$$
\begin{array}{l}
\textit{prog} ::= \textit{decl}^{*}\ e \\
\textit{decl} ::= \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{\textit{field}}\ \ \overline{\textit{meth}}\ \ \overline{\textit{binding}}\ \}\ \mid \texttt{event}\ p\ \{\ \overline{\textit{form}}\ \} \\
\textit{field} ::= c\ f\,\texttt{;} \\
\textit{meth} ::= c\ m\ \texttt{(}\ \overline{\textit{form}}\ \texttt{)\{}\ e\ \texttt{\}} \\
t ::= c \mid \texttt{void} \\
\textit{binding} ::= \texttt{when}\ p\ \texttt{do}\ m\ \texttt{;} \\
\textit{form} ::= c\ \textit{var},\ \text{ where } \textit{var} \neq \texttt{this} \\
e ::= \texttt{new}\ c\texttt{()} \mid \textit{var} \mid \texttt{null} \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid e\texttt{.}f \mid e\texttt{.}f\ \texttt{=}\ e \mid \texttt{cast}\ c\ e \mid \textit{form}\ \texttt{=}\ e\ \texttt{;}\ e \mid e\ \texttt{;}\ e \\
\qquad \mid \texttt{register(}e\texttt{)} \mid \texttt{announce}\ p\ \texttt{(}\overline{e}\ \texttt{)}\ \texttt{;}\ e
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{where} & & \\
c, d & \in & \mathcal{C},\ \text{the set of class names} \\
p & \in & \mathcal{P},\ \text{the set of event type names} \\
f & \in & \mathcal{F},\ \text{the set of field names} \\
m & \in & \mathcal{M},\ \text{the set of method names} \\
\textit{var} & \in & \{\texttt{this}\} \cup \mathcal{V}, \mathcal{V} \text{ is the set of variable names}
\end{array}
$$

Figure 3.1 Pāñini's abstract syntax, based on [50, 13].

These context declarations specify the types and names of reflective information exposed by conforming events. An example is given in Figure 1.2 on lines 1-3 where `event GenAvailable` has one context variable `Generation g` that 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. The two top-level declaration forms, classes and event type declarations, may not be nested.

Like Eos [51], classes in Pāñini may also contain binding declarations. A binding declaration (*BindingDecl*) mainly consists of two parts: an event type name (*Type*) and a method name (*Identifier*). For example, in Figure 1.2 on line 10 the class `CrossOver` declares a binding such that the `cross` method is invoked whenever an `event` of type `GenAvailable` is announced. We call such methods *handler methods* and they may run concurrently with other handler methods for the same event.

Pāñini's New Statements. Pāñini has all the standard object-oriented expressions and statements as in Java. New to Pāñini is the registration statement (*RegisterStmt*) and (*AnnounceStmt*). Like II languages and Ptolemy [50], a module in Pāñini can express interest in events, e.g., to implement the observer design pattern [24]. Just like II languages, where one has to write a statement for registering a handler with each event in a set, and similar to Ptolemy [50], such modules run registration statements. Examples are shown on lines 7, 23, 38 and 44 in Figure 1.2. The example on line 7 registers the `this` object to receive notification when events of type `GenAvailable` are signaled.

3.2 Concurrency in Pāñini

The `announce` statement enables concurrency in Pāñini. The statement `announce p (` *Expr** `) ;` signals an event of type *p*, which may run any handler methods that are applicable to *p* *asynchronously*, and waits for the handlers to finish. In Figure 1.2, the body of the `cross` method contains an `announce` statement on line 18. On evaluation of the announce statement, Pāñini 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.

The evaluation of the announce statement 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 statement 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`.

3.3 Pāñini's Handler Conflict Detection Scheme

Pāñini uses static effect computation [60] and a dynamic conflict detection scheme to compute a schedule for execution of handlers that maximizes concurrency while ensuring a deterministic semantics of programs. This is similar to Jade [53], where the implementation tries to discover concurrency. But unlike Jade, we do not require effect annotations. Pāñini's compiler generates code to compute the potential effect of all handlers. At runtime, when a handler registers with an event, Pāñini's runtime uses these statically computed effects to decide the execution schedule of handlers.

Effects of a Method. The effects of a method are modeled as a set that may contain four kinds of effects: 1) read effect: a class and its field that may be read; 2) write effect: a class and its field that may be written; 3) announce effect: an event that may be announced by the method; 4) register effect: whether this method may evaluate a `register` statement. These sets are generated for each method in the program and inserted in the generated code as synthetic methods. For library methods, their effects are computed by analyzing their bytecode and inserted directly at call-sites.

Detecting Dependencies between Handlers. When a `register` statement is run with a handler as argument, dependence between this handler and already registered handlers for that event is computed by comparing their effects. Two handlers may have *read-write*, *write-write* or *register-announce* dependencies.

Suppose the currently registering handler is h_r and h_i is in the sequence of already registered handlers. Handlers h_r and h_i may be register-announce dependent if h_i announces an event for which h_r registers a handler or vice versa. The handler h_r is read-write dependent on h_i if h_r's reads conflict with h_i's writes, or h_r's writes conflict with h_i's reads or writes. Two effect sets conflict, if they share an element. That is because, in the deterministic semantics, h_r should view the changes by h_i, while h_r's changes are invisible to h_i, neither should the changes of h_r be overwritten by the changes of h_i. We illustrate via an example in Figures 3.2-3.4.

Handlers	Reads	Writes	Registers	Announces
A	{Account.balance}	$\emptyset$	$\emptyset$	{`Ev`}
B	{Account.id}	$\emptyset$	$\emptyset$	$\emptyset$
C	$\emptyset$	{Account.balance}	$\emptyset$	$\emptyset$

Figure 3.2 Initial effects of three handlers

Assume that there are three handlers A, B and C, with their initial effects shown in Figure 3.2. 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's writes conflict with handler A's reads, as discussed above.

Notice that a handler h could also announce an event, say p. Then the read/write set of h could be enlarged over time, because new handlers for p may register later and the effects of these new handlers should propagate to h. Pāñini does these updates automatically when new handlers register for a certain event. To enable this, subjects are formed into a list for an event. Thus, when a handler registers, its changes are passed to these subjects, and these subjects merge the changes and recursively pass changes to other events when necessary. This continues until a fixpoint is reached (no more effects are added to the subjects). For example, in Figure 3.3 notice that handler A may announce events of type `Ev`. Thus after handler B registers, the effect set of handler A becomes the union of effect sets of handlers A and B.

Handlers	Reads	Writes	Registers	Announces
A	{Account.balance, Account.id}	$\emptyset$	$\emptyset$	{`Ev`}
B	{Account.id}	$\emptyset$	$\emptyset$	$\emptyset$
C	$\emptyset$	{Account.balance}	$\emptyset$	$\emptyset$

Figure 3.3 Effects after handler A and handler B have registered.

Finally, in Figure 3.4, the effect set of handler A becomes the union of effect sets of all the three handlers.

Handlers	Reads	Writes	Registers	Announces
A	{Account.balance, Account.id}	{Account.balance}	$\emptyset$	{`Ev`}
B	{Account.id}	$\emptyset$	$\emptyset$	$\emptyset$
C	$\emptyset$	{Account.balance}	$\emptyset$	$\emptyset$

Figure 3.4 Effects after all three handlers have registered.

Handlers' Hierarchy. Pāñini groups handlers into hierarchies, based on handler dependencies. In the first level of the hierarchy, 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, handler C conflicts with handler A (discussed previously). 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 above) and in response to this, Pāñini will reorder the hierarchy dynamically. Thus, the example above will have a two level hierarchy, with handlers A and B in the first level, while, handler C in the second.

Event Registration. When a handler, say h, registers with event p, we first propagate its effects to the subjects of p. Then the dependencies between h and the previous registered handlers are computed based on the effect set. After dependencies are calculated, the handler is put into a proper level of the hierarchy. In Figure 3.3 and Figure 3.4, since, handler A may announce event type `Ev`, the effect sets of handler B and handler C are propagated to handler A (as a subject). 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's writes conflict with handler A's reads, it is put in the second level.

Event Announcement and Task Scheduling Algorithm. When a subject signals an event, Pāñini executes the handlers in the first level concurrently (the subject itself blocks until all handlers are finished). 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, since handlers A and B are both in the first level, they will run in parallel. Once they are completed, handler C will run. If any of the handlers also announce an event, the handlers for that event will be scheduled, according to their conflict sets. Announce statements do not return until after all the handlers associated with the event are finished. This ensures correct synchronization for any state changes made by the handlers.

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 effect analysis is amortized over event announcements.

3.4 Properties of Pāñini's Design

Pāñini does not have locks so it is deadlock free. It uses automatic conflict detection that ensures race freedom and guaranteed deterministic semantics. Chapter 6 has formal details and proofs of these properties. Its design, does not offer these guarantees if programmers use explicit locking and threads in the underlying Java language in a manner that creates deadlocks and data races.

CHAPTER 4. Pāñini's Static Semantics and Effect System

In this chapter, we present Pāñini's type system. It is builds on our previous work on the Ptolemy [50] and Eos [52] languages. The Ptolemy language did not use the effect system [39] to facilitate concurrency. Other type systems, for example [40], which studied the effects of computations, do not provide a static semantics for event-based concurrent programs. Thus, addressing these problems is necessary to formalize Pāñini's static semantics.

4.1 Type and Effect Attributes

Type checking uses the attributes defined in Figure 4.1. Compared to Ptolemy [50] new to Pāñini is its effect system. For example, the type attributes for expressions are represented as (t, ρ), the type of the expression (t) and its effect set (ρ).

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

Figure 4.1 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: an event that a certain expression may announce; 4) register effect and 5) new object. The interference between the effects is shown in Figure 4.2.

Effects	`read`	`write`	`ann`	`reg`	`create`
`read`	$\times$	$\surd$	$\times$	$\surd$	$\times$
`write`	$\surd$	$\surd$	$\times$	$\surd$	$\times$
`ann`	$\times$	$\times$	$\times$	$\surd$	$\times$
`reg`	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
`create`	$\times$	$\times$	$\times$	$\times$	$\times$

Figure 4.2 Effect Interference. $\surd$: conflicts, $\times$: no conflicts

4.2 Effect Interference

Read effects do not interfere with any other read effects. Write effects conflict with either another read or write effect accessing 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 registration, a handler could introduce unforeseen read and write effects and thus complicate the interference.

New object effect will not interfere with any other effects. 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 cannot 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, Pāñini could safely remove the read/write effect of any newly created object and thus reduce false interferences to a considerable extent.

4.3 Type Checking Rules

The type checking rules are shown in Figures 4.3, 4.6 and 4.7. The notation $\nu' <: \nu$ means ν' is a subtype of ν. It is the reflexive-transitive closure of the declared subclass relationships [50]. We state the type checking rules, using a fixed class table (list of declarations CT) as in Clifton's work [13, 14]. The class table can be thought of as an implicit inherited attribute used by the rules and auxiliary functions. We require that top-level names in the program are distinct and that the inheritance relation on classes is acyclic. The typing rules for expressions use a simple type environment, Π, which is a finite partial mapping from locations *loc* or variable names *var* to a type.

4.3.1 Rules for Declarations

The rules for top-level declarations are fairly standard and are shown in Figures 4.3.

The (T-Program) rule says that the entire program type checks if all the declarations type check and the expression e has any type t and any effect ρ.

The (T-Event) rule says that an event declaration type checks, if all the types of all the fields are declared properly. The auxiliary function *isType* (shown in Figure 4.4), looks at the class table to check if a type has been defined or not.

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 ρ. Also u

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

(T-Event)
$$\frac{(\forall (t_i\ var_i) \in \overline{t\ var\texttt{;}} :: isType(t_i))}{\vdash \texttt{event}\ \ p\ \{\overline{t\ var\texttt{;}}\} : \mathrm{OK}}$$

(T-Class)
$$\frac{validF(\overline{t\ f}, d) \quad isClass(d) \quad (\forall meth_j \in \overline{meth} \ :: \ \vdash \ meth_j : (\theta_j, \rho_j)\ in\ C) \quad (\forall b \in \overline{binding} \ :: \ \vdash \ b : \mathrm{OK\ in}\ \ C)}{\vdash \ \texttt{class}\ \ c\ \texttt{extends}\ d\{\overline{t\ f;}\ \ \overline{meth}\ \ \overline{binding}\} : \mathrm{OK}}$$

(T-Method)
$$\frac{isClass(t) \quad (\forall i \in \{1..n\} :: isClass(t_i)) \quad var_1 : t_1, \ldots, var_n : t_n, \texttt{this} : c \vdash e : (u, \rho) \quad u <: t \quad override(m, c, (t_1 \times \ldots \times t_n \rightarrow t, \rho))}{\vdash \ t\ m(t_1\ var_1, \ldots, t_n\ var_n)\{e\} : (t_1 \times \ldots \times t_n \rightarrow t, \rho)\ \mathrm{in}\ c}$$

(T-Binding)
$$\frac{CT(p) = \texttt{event}\ \ p\ \{t'_1\ var'_1, \ldots, t'_m\ var'_m\} \quad (c_1, t\ m(\overline{t\ var})\{e\}, \rho) = findMeth(c, m) \quad \pi = \{var'_1 : t'_1\ , \ldots, var'_m : t'_m\} \quad (\forall\ t_i\ var_i \in \overline{t\ var}\ \ ::\ \ \pi(var_i) <:\ t_i)}{\vdash \ \texttt{when}\ p\ \texttt{do}\ m\ \ : \mathrm{OK\ in}\ c}$$

Figure 4.3 Type and Effect rules for declarations [13, 14, 50].

$$isType(t) = (t \in dom(CT) \wedge CT(t) = \texttt{class}\ \ t \ldots)$$

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

Figure 4.4 Auxiliary functions used in type rules[49].

is a subtype of t. This rule also uses an auxiliary function *override*, defined in Figure 4.5. In addition to standard conditions, this function enforces that the effect of an overriding method is the subset of the effect of overridden method[1].

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.

[1]In practice, we enlarge the effect set of the method in the super class such that the effect of the overriding method is a subset of its super class.

Valid method overriding:

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ \ c\ \texttt{extends}\ d\ \{\mathit{field}^*\ \mathit{meth}_1 \ldots \mathit{meth}_p\ \mathit{bind}_1 \ldots \mathit{bind}_q\} \\ \nexists i \in \{1..p\} \cdot \mathit{meth}_i = t\ m(t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\{e\} \qquad \mathit{override}(m, d, (t_1 \times \ldots \times t_n \to t, \rho)) \end{array}}{\mathit{override}(m, c, t_1 \times \ldots \times t_n \to t, \rho)}$$

$$\frac{\mathit{methodType}(d, m) = (t_1 \times \ldots \times t_n \to t, \rho') \qquad \rho \subseteq \rho'}{\mathit{override}(m, d, (t_1 \times \ldots \times t_n \to t, \rho))}$$

$$\mathit{override}(m, \mathit{Object}, t_1 \times \ldots \times t_n \to t, \rho)$$

Figure 4.5 Auxiliary functions used in type rules[13, 14].

4.3.2 Rules for Expressions

The type rules for the expressions are shown in Figure 4.6 and Figure 4.7. Most rules for typing expressions are straightforward.

$$\text{(T-NEW)}\ \frac{\mathit{isClass}(c)}{\Pi \vdash \texttt{new}\ c() : (c, \{\texttt{create}\ c\})} \qquad \text{(T-CAST)}\ \frac{\mathit{isType}(t) \qquad \Pi \vdash e : (u, \rho)}{\Pi \vdash \texttt{cast}\ t\ e : (t, \rho)} \qquad \text{(T-SEQUENCE)}\ \frac{\Pi \vdash e_1 : (t_1, \rho) \qquad \Pi \vdash e_2 : (t_2, \rho')}{\Pi \vdash e_1; e_2 : (t_2, \rho \cup \rho')}$$

$$\text{(T-VAR)}\ \frac{\Pi(\mathit{var}) = (t, \rho)}{\Pi \vdash \mathit{var} : (t, \rho)} \qquad \text{(T-NULL)}\ \frac{\mathit{isClass}(c)}{\Pi \vdash \texttt{null} : (c, \emptyset)}$$

$$\text{(T-CALL)}\ \frac{\begin{array}{c} (c_1, t\ m(t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\{e_{n+1}\}, \rho) = \mathit{findMeth}(c_0, m) \\ c_0 <: c_1 \qquad \Pi \vdash e_0 : (c_0, \rho_0) \qquad (\forall\ i \in \{1..n\}\ ::\ \Pi \vdash e_i : (u_i, \rho_i)\ \wedge\ u_i <: t_i) \end{array}}{\Pi \vdash e_0.m(e_1, \ldots, e_n) : (t, \rho \cup \bigcup_{i=1}^{n} \rho_i\ \cup \rho_0)}$$

Figure 4.6 Type and Effect rules for expressions[13, 14, 50].

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-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-Var) rule checks that var is in the environment.

The (T-Null) rule says that the null expression will type check and has no 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 `methodBody` in Clifton's work [13, 14], except that here it also returns the effect set of the method.

$$\text{(T-GET)}\quad \frac{\Pi \vdash e : (c, \rho) \qquad \rho \neq \{\texttt{create}\ c\} \qquad \mathit{fieldsOf}(c)(f) = t}{\Pi \vdash e.f : (t, \rho \cup \{\texttt{read}\ \ c\ f\})}$$

$$\text{(T-GET-LOCAL)}\quad \frac{\Pi \vdash e : (c, \{\texttt{create}\ c\}) \qquad \mathit{fieldsOf}(c)(f) = t}{\Pi \vdash e.f : (t, \{\})}$$

$$\text{(T-YIELD)}\quad \frac{\Pi \vdash e : (t, \rho)}{\Pi \vdash \texttt{yield}\ \ e : (t, \rho)}$$

$$\text{(T-DEFINE)}\quad \frac{\mathit{isType}(t) \qquad \Pi \vdash e_1 : (t_1, \rho) \qquad \Pi, \mathit{var} : (t, \rho) \vdash e_2 : (t_2, \rho') \qquad t_1 <: t}{\Pi \vdash t\ \mathit{var} = e_1; e_2 : (t_2, \rho \cup \rho')}$$

$$\text{(T-REGISTER)}\quad \frac{\Pi \vdash e : (t, \rho) \qquad \mathit{isClass}(t)}{\Pi \vdash \texttt{register(}e\texttt{)} : (t, \rho \cup \{\texttt{reg}\ \})}$$

$$\text{(T-SET)}\quad \frac{\Pi \vdash e : (c, \rho) \qquad \rho \neq \{\texttt{create}\ c\} \qquad \mathit{fieldsOf}(c)(f) = t \qquad \Pi \vdash e' : (t', \rho') \qquad t' <: t}{\Pi \vdash e.f = e' : (t', \rho \cup \rho' \cup \{\texttt{write}\ \ c\ f\})}$$

$$\text{(T-SET-LOCAL)}\quad \frac{\Pi \vdash e : (c, \{\texttt{create}\ c\}) \qquad \mathit{fieldsOf}(c)(f) = t \qquad \Pi \vdash e' : (t', \rho') \qquad t' <: t}{\Pi \vdash e.f = e' : (t', \rho \cup \rho')}$$

$$\text{(T-ANNOUNCE)}\quad \frac{CT(p) = \texttt{event}\ \ p\ \{t_1\ \mathit{var}_1\texttt{;} \ldots t_n\ \mathit{var}_n\texttt{;}\} \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (u_i, \rho_i)\ \wedge\ u_i <: t_i) \qquad \Pi \vdash e : (t, \rho)}{\Pi \vdash \texttt{announce}\ p\ \texttt{(}e_1, \ldots, e_n\texttt{);}e : (t, \{\texttt{ann}\ \ p\} \cup \bigcup_{i=1}^{n} \rho_i\ \cup\ \rho)}$$

Figure 4.7 Type and Effect rules for expressions that generate new effects.

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 Pāñini knows that the object expression is pointing to a newly created object and thus the read effect is redundant and deleted.

The (T-Yield) rule says that a `yield` expression has the same type and same effect as the expression e.

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-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_2 is a subtype of the type of the field of the class. The effects will be the union of the effects of its two subexpressions plus one `write` effect, 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 field's 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.

```
event GenAvailable {
  Generation g;
}
class Generation{}
class LastGen {
  Generation g;
  LastGen init(){
    register(this);
    this.g = new Generation();
    this
  }
  when GenAvailable do record;
  Generation record(Generation g) {
    this.g = g; this.g
  }
}
LastGen ng = new LastGen().init();
announce GenAvailable(new Generation());
ng.g
```

Figure 4.8 An Pāñini program

Figure 4.8 shows an example program. The program defines one event type `GenAvailable`, which has a context variable `g` of type `Generation`. There are two classes defined, namely `Generation` and `LastGen`. The class has two methods (`init` and `record`) and one event binding, which says that when event of type `GenAvailable` is fired, it will execute the method `record`. Following the declarations, there is an expression, which could be thought of as a main method of a Java program.

$$\text{(T-Event)}\quad \frac{\mathit{isType}(\mathit{Generation})}{\vdash \texttt{event}\ \ \mathit{GenAvailable}\ \{\mathit{Generation}\ g;\} : \mathrm{OK}} \qquad \text{(T-Class)}\quad \frac{\mathit{isClass}(\mathit{Object})}{\vdash\ \texttt{class}\ \ \mathit{Generation}\ \texttt{extends}\ \mathit{Object}\{\} : \mathrm{OK}}$$

Figure 4.9 Type System Example: checking `event GenAvailable` and `class Generation`.

Clearly, the event type `GenAvailable` and the class `Generation` type checks, following from Figure 4.9.

$$\text{(T-Register)}\quad \frac{}{\begin{array}{c}\texttt{this} : \mathit{LastGen} \vdash \\ \texttt{register(this)} : \\ (\mathit{LastGen}, \{\texttt{reg}\ \})\end{array}}$$

$$\text{(T-set)}\quad \frac{\begin{array}{c}\texttt{this} : \mathit{LastGen} \vdash \texttt{this} : (\mathit{LastGen}, \{\}) \\ \mathit{fieldsOf}(\mathit{LastGen})(g) = \mathit{Generation} \qquad \mathit{isClass}(\mathit{Generation}) \\ \texttt{this} : \mathit{LastGen} \vdash \mathit{new\ Generation}() : (\mathit{Generation}, \{\texttt{create}\ \mathit{LastGen}\})\end{array}}{\begin{array}{c}\texttt{this} : \mathit{LastGen} \vdash \texttt{this}.g = \mathit{new\ Generation}() : \\ (\mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \mathit{LastGen}\})\end{array}}$$

$$\text{(T-sequence)}\quad \frac{\begin{array}{c}\Pi \vdash \texttt{register(this)} : (\mathit{LastGen}, \{\texttt{reg}\ \}) \\ \Pi \vdash \texttt{this}.g = \mathit{new\ Generation}() : (\mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \mathit{Generation}\})\end{array}}{\begin{array}{c}\texttt{this} : \mathit{LastGen} \vdash \texttt{register(this)}; \texttt{this}.g = \mathit{new\ Generation}(); \texttt{this} : \\ (\mathit{LastGen}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \mathit{Generation}, \texttt{reg}\ \})\end{array}}$$

$$\text{(T-Method)}\quad \frac{\begin{array}{c}\mathit{isClass}(\mathit{Generation}) \qquad \texttt{this} : \mathit{LastGen} \vdash e : (\mathit{LastGen}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \mathit{Generation}, \texttt{reg}\ \}) \\ \mathit{override}(\mathit{init}, \mathit{LastGen}, (\rightarrow \mathit{LastGen}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \mathit{Generation}, \texttt{reg}\ \}))\end{array}}{\vdash\ \mathit{Generation\ init}()\{e\} : (\rightarrow \mathit{LastGen}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \mathit{Generation}, \texttt{reg}\ \})\ \mathit{inLastGen}}$$

Figure 4.10 Type System Example: checking method `init`.

The method `init` in the class `Last` type checks, following from the last rule in Figure 4.10. This method consists of three expressions: a register expression, a set expression and a var expression.

The method `record` in Figure 4.11 type checks similarly.

The class `LastGen` in Figure 4.12 type check, because all the two of its methods type checks in the previous discussion, its field has a proper type and finally the only event binding declaration type checks. The event binding type checks, because the event type was properly

$$\text{(T-SET)}\quad \frac{\begin{array}{c}\texttt{this} : \mathit{LastGen} \vdash \texttt{this} : (\mathit{LastGen}, \{\}) \qquad \mathit{isClass}(\mathit{Generation}) \\ \mathit{fieldsOf}(\mathit{LastGen})(g) = \mathit{Generation}\end{array}}{\{\texttt{this} : \mathit{LastGen}, g : \mathit{Generation}\} \vdash \texttt{this}.g = g : (\mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g\})}$$

$$\text{(T-SEQUENCE)}\quad \frac{\begin{array}{c}\{\texttt{this} : \mathit{LastGen}, g : \mathit{Generation}\} \vdash \texttt{this}.g = g : (\mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g\}) \\ \{\texttt{this} : \mathit{LastGen}, g : \mathit{Generation}\} \vdash \texttt{this}.g : (\mathit{Generation}, \{\texttt{read}\ \ c\ g\})\end{array}}{\{\texttt{this} : \mathit{LastGen}, g : \mathit{Generation}\} \vdash \texttt{this}.g = g; \texttt{this}.g : (\mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{read}\ \ c\ g\})}$$

$$\text{(T-METHOD)}\quad \frac{\begin{array}{c}\mathit{isClass}(\mathit{Generation}) \qquad \mathit{override}(\mathit{record}, \mathit{LastGen}, (\mathit{Generation} \to \mathit{Generation}, \mathit{Generation})) \\ g : \mathit{Generation}, \texttt{this} : c \vdash \texttt{this}.g = g; \texttt{this}.g : (\mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{read}\ \ c\ g\})\end{array}}{\begin{array}{c}\vdash\ \mathit{Generation}\ \mathit{record}(\mathit{Generation}\ g)\{\texttt{this}.g = g; \texttt{this}.g\} : \\ (\mathit{Generation} \to \mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{read}\ \ c\ g\})\ \mathit{in}\ \mathit{LastGen}\end{array}}$$

Figure 4.11 Type System Example: checking method `record`.

$$\text{(T-BINDING)}\quad \frac{\begin{array}{c}\mathit{CT}(\mathit{GenAvailable}) = \texttt{event}\ \ \mathit{GenAvailable}\ \{\mathit{Generation}\ g\} \\ (\mathit{LastGen}, \mathit{Generation}\ \mathit{record}(\mathit{Generation}\ g)\{\ldots\}, \rho) = \mathit{findMeth}(\mathit{LastGen}, \mathit{record}) \qquad \mathit{Generation} <: \mathit{Generation}\end{array}}{\vdash\ \texttt{when}\ \mathit{GenAvailable}\ \texttt{do}\ \mathit{record}\ : \text{OK in } \mathit{LastGen}}$$

$$\text{(T-CLASS)}\quad \frac{\begin{array}{c}\mathit{validF}(\mathit{Generation}\ g, \mathit{Object}) \\ \mathit{LastGen}\ \mathit{init}()\{\ldots\} : (\to \mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{create}\ \ \mathit{LastGen}, \texttt{reg}\ \}) \ \in \mathit{LastGen} \\ \mathit{Generation}\ \mathit{record}(\mathit{Generation}\ g)\{\ldots\} : (\mathit{Generation} \to \mathit{Generation}, \{\texttt{write}\ \ \mathit{LastGen}\ g, \texttt{read}\ \ c\ g\})\ \mathit{in}\ \mathit{LastGen} \\ \texttt{when}\ \mathit{GenAvailable}\ \texttt{do}\ \mathit{record}\ : \text{OK in } \mathit{LastGen}\end{array}}{\begin{array}{c}\vdash\ \texttt{class}\ \ \mathit{LastGen}\ \texttt{extends}\ \mathit{Object}\{\mathit{Generation}\ g; \\ \mathit{LastGen}\ \mathit{init}()\{\ldots\} \qquad \mathit{Generation}\ \mathit{record}(\mathit{Generation}\ g)\{\ldots\} \\ \texttt{when}\ \mathit{GenAvailable}\ \texttt{do}\ \mathit{record} : \text{OK}\end{array}}$$

Figure 4.12 Type System Example: checking `class Generation`.

defined and the handler method `record` type checks and all the parameter types are subtypes of the context variables.

Eventually, the program expression type checks as is shown in Figure 4.13. This expression first defines a local variable `ng`, which is initiated to a newly created object. The object of the class `LastGen` was properly declared. An event of type `GenAvailable` is fired after that. The entire program type checks because all the declarations (classes and event type) and the main expression type check, as discussed in the previous paragraphs.

(T-Call)

$$\frac{\begin{array}{c}(LastGen, LastGen\ init\texttt{()}\{e\}, \{\texttt{write}\ \ LastGen\ g, \texttt{create}\ \ Generation, \texttt{reg}\ \})\\ = findMeth(LastGen, init) \qquad \vdash \texttt{new}\ \ LastGen() : (LastGen, \{\texttt{create}\ \ LastGen\})\end{array}}{\begin{array}{c}\vdash \texttt{new}\ \ LastGen().init() :\\ (LastGen, \{\texttt{write}\ \ LastGen\ g, \texttt{create}\ \ Generation, \texttt{reg}\ , \texttt{create}\ \ LastGen\})\end{array}}$$

(T-Announce)

$$\frac{\begin{array}{c}CT(GenAvailable) = \texttt{event}\ \ GenAvailable\ \{Generation\ g\texttt{;}\}\\ ng : (LastGen, \ldots) \vdash ng.g : (Generation, \{\texttt{read}\ \ LastGen\ g\})\\ ng : (LastGen, \ldots) \vdash \texttt{new}\ \ Generation() : (Generation, \{\texttt{create}\ \})\end{array}}{\begin{array}{c}ng : (LastGen, \ldots) \vdash \texttt{announce}\ GenAvailable\ \texttt{(new}\ \ Generation()\texttt{)}\,\texttt{;}ng.g :\\ (Generation, \{\texttt{ann}\ \ p, \texttt{read}\ \ LastGen\ g, \texttt{create}\ \})\end{array}}$$

(T-define)

$$\frac{\begin{array}{c}\vdash \texttt{new}\ \ LastGen().init() : (LastGen, \{\texttt{write}\ \ LastGen\ g, \texttt{create}\ \ Generation, \texttt{reg}\ , \texttt{create}\ \ LastGen\})\\ ng : (LastGen, \{\texttt{write}\ \ LastGen\ g, \texttt{create}\ \ Generation, \texttt{reg}\ , \texttt{create}\ \ LastGen\}) \vdash\\ \texttt{announce}\ \ldots : (Generation, \{\texttt{ann}\ \ p, \texttt{read}\ \ LastGen\ g, \texttt{create}\ \ Generation\})\end{array}}{\begin{array}{c}\vdash LastGen\ ng = \texttt{new}\ \ LastGen().init(); \texttt{announce}\ \ldots : (t_2,\\ \{\texttt{write}\ \ LastGen\ g, \texttt{create}\ \ Generation, \texttt{reg}\ , \texttt{create}\ \ LastGen, \texttt{ann}\ \ p, \texttt{read}\ \ LastGen\ g\})\end{array}}$$

Figure 4.13 Type System Example: checking the main expression.

CHAPTER 5. Pāñini's Operational Semantics

Here we give a small-step operational semantics for Pāñini. Reactor [56] and other works have studied the concurrent pattern for distributed event-based systems. These works [21, 31] did not provide a formal semantics for their system, were sequential, or only assumed disjoint address space between concurrent tasks. Addressing these problems is necessary to formalize Pāñini's operational semantics. On the other hand, Pāñini provides a dynamic effect system, with which Pāñini could more accurately detect conflict between concurrent tasks.

5.1 Added syntax, domains, and evaluation contexts

Added Syntax:

$e \quad ::= \; loc \mid \texttt{yield}\ e \mid \texttt{NullPointerException} \mid \texttt{ClassCastException}$

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

Figure 5.1 Added syntax based on [13, 50].

Intermediate Expressions. The expression semantics rely on four expressions that are not part of Pāñini's surface syntax as shown in Figure 5.1. 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 5.2) 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.

(NCall)
$$\begin{array}{l}\langle\langle \mathbb{E}[\texttt{null}.m(v_1,\ldots,v_n)],\tau\rangle + \psi,\mu,\gamma\rangle \\ \hookrightarrow \langle\langle \mathit{NullPointerException},\tau\rangle + \psi,\mu,\gamma\rangle\end{array}$$

(NGet)
$$\begin{array}{l}\langle\langle \mathbb{E}[\texttt{null}.f],\tau\rangle + \psi,\mu,\gamma\rangle \\ \hookrightarrow \langle\langle \mathit{NullPointerException},\tau\rangle + \psi,\mu,\gamma\rangle\end{array}$$

(NSet)
$$\begin{array}{l}\langle\langle \mathbb{E}[\texttt{null}.f = v],\tau\rangle + \psi,\mu,\gamma\rangle \\ \hookrightarrow \langle\langle \mathit{NullPointerException},\tau\rangle + \psi,\mu',\gamma\rangle\end{array}$$

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

Figure 5.2 Operational semantics of expressions that produce exceptions.

Domains. The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 5.3.

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

Σ	$::=$	$\langle \psi,\mu,\gamma\rangle$	"Program Configurations"
ψ	$::=$	$\langle e,\tau\rangle + \psi \mid \bullet$	"Task Configurations"
τ	$::=$	$\langle n, \{n_k\}_{k\in K}\rangle$ **where** $n_k \in \mathcal{N}$ and K is finite	"Task Local Data"
μ	$::=$	$\{loc \mapsto o\} + \mu \mid \bullet$	"Stores"
v	$::=$	$\texttt{null} \mid loc$	"Values"
o	$::=$	$[c.F]$	"Object Records"
F	$::=$	$\{f_k \mapsto v_k\}_{k\in K}$, **where** K is finite	"Field Maps"
γ	$::=$	$loc + \gamma \mid \bullet$	"Subscriber List"

Figure 5.3 Domains used in the semantics, based on [13, 50].

A configuration consists of a task queue ψ, a global store μ, and a global subscriber list γ. The store μ is a mapping from locations (loc) to objects (o). The subscriber list γ consists of a set of receiver objects for handler methods.

The task queue ψ consists of an ordered list of task configurations $\langle e,\tau\rangle$. This configuration consists of an expression e running in that task and the corresponding task local data (τ). 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 it depends on are finished.

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

Evaluation contexts:

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

Figure 5.4 Evaluation contexts used in the semantics, based on [13, 50].

Evaluation Contexts. We present the semantics as a set of evaluation contexts $\mathbb{E}$ (Figure 5.4) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context [63]. 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 $\langle\langle e, \langle 0, \emptyset\rangle\rangle, \bullet, \bullet\rangle$.

5.2 Semantics for Object-oriented Expressions

The rules for OO expressions are given in Figure 5.5. These are mostly standard and adopted from the work of Rajan and Leavens [50], and Clifton's work [13, 14].

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

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

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

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

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

(Call)
$$\frac{(c', t, m(t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\{e\}, \rho) = \mathit{findMeth}(c, m) \qquad [c.F] = \mu(\mathit{loc}) \qquad e' = [\texttt{this}/\mathit{loc}, \mathit{var}_1/v_1, \ldots, \mathit{var}_n/v_n]e}{\langle\langle \mathbb{E}[\mathit{loc}.m(v_1, \ldots, v_n)], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\ e'], \tau\rangle + \psi, \mu, \gamma\rangle}$$

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

Figure 5.5 Semantics of object-oriented expressions in Pāñini, based in part on [50, 13, 14]

One difference stems from the concurrency and store models in Pāñini. 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 nondeterministic and thus Pāñini models this by nondeterministically 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 [13, 14]). It uses dynamic dispatch, which starts from the dynamic class (c) of the record, and may look up the super class of the object if needed. The method body is to be evaluated with the arguments 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.

5.3 Semantics for Yielding Control

In Pāñini'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 5.6.

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

$$(\textsc{Yield-Done})\quad \langle \langle \mathbb{E}[\texttt{yield}\ e], \tau \rangle + \bullet, \mu, \gamma \rangle \hookrightarrow \langle \langle \mathbb{E}[e], \tau \rangle + \bullet, \mu, \gamma \rangle$$

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

Figure 5.6 Semantics of yielding control in Pāñini

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 5.7). 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.

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

Figure 5.7 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.

5.4 Semantics for Event registration

We now describe the semantics for subscribing to an event (Figure 5.8).

$$\textsc{(Multi-Register)}\quad \frac{loc \in \gamma}{\langle\langle \mathbb{E}[\texttt{register}(loc)], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[loc], \tau\rangle + \psi, \mu, \gamma\rangle}$$

$$\textsc{(Register)}\quad \frac{loc \notin \gamma}{\langle\langle \mathbb{E}[\texttt{register}(loc)], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[loc], \tau\rangle + \psi, \mu, loc + \gamma\rangle}$$

Figure 5.8 Semantics of Registration

The (Multi-Register) rule is applied when the handler has already registered previously and thus the configuration does not change. Pāñini 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 Pāñini safely puts this record at the front of the queue.

5.5 Semantics for announcing an event

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

Figure 5.9 Semantics of Announcement

The semantics for signaling events is shown in Figure 5.9.

The (Announce) rule takes the relevant event declaration from the program's list of declarations and creates a list of actual parameters (ν). This list of actual parameters (ν) is used by the auxiliary function *spawn* shown in Figure 5.10 (with other helper functions in Figure 5.12 and Figure 5.11). 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 5.11) 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 (γ) for applicable handlers (using auxiliary functions *hfind*, *hmatch*, and *match*). Auxiliary functions *buildconfs* (Fig-

$spawn(p, \psi, \gamma, \nu, \mu) = buildconfs(H, \psi, \nu, \bullet, \gamma, \mu)$
  where $H = hfind(\gamma, p, \mu)$ **and** CT is the program's list of declarations

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

$hmatch(c, p, \bullet) = \bullet$
$hmatch(c, p, (\texttt{event}\ \ p\{\dots\}) + CT') = hmatch(c, p, CT')$
$hmatch(c, p, (\texttt{class}\ c' \dots) + CT') = hmatch(c, p, CT')$ **where** $c \neq c'$
$hmatch(c, p, ((\texttt{class}\ c\ \texttt{extends}\ d\ \dots\ binding_1 \dots binding_n) + CT'))$
$= excl(match((binding_n + \dots + binding_1), p), hmatch(d, p, CT))$
  where $excl(\bullet, H) = H$ **and** $excl(e, H) = e$

$match(\bullet, p) = \bullet$
$match((\texttt{when}\ p'\ \texttt{do}\ m) + B, p) = match(H, p)$ **where** $p' \neq p$
$match((\texttt{when}\ p'\ \texttt{do}\ m) + B, p) = m$

Figure 5.10 Functions for Creating Task Configurations.

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

$max(\bullet, id) = id$
$max(\langle e', \langle id', I\rangle\rangle + \psi, id) = max(\psi, id)$ **where** $id' < id$
$max(\langle e', \langle id', I\rangle\rangle + \psi, id) = max(\psi, id')$ **where** $id' > id$

$getE(\bullet, \gamma, \mu) = \bullet$
$getE(\langle loc, m\rangle + H, \gamma, \mu) = concat(update(\rho, \gamma, \mu), getE(H, \gamma, \mu))$
  where $loc = [c.F]$ **and** $(c', t, m \dots, (\dots, \rho)$ in $c') = findMeth(c, m)$

$diff(\bullet, \rho) = true$
$diff(\epsilon + \rho', \rho) = diff(\rho', \rho)$ **where** $true = differ(\epsilon, \rho)$
$diff(\epsilon + \rho', \rho) = false$ **where** $false = differ(\epsilon, \rho)$

$differ(\epsilon, \bullet) = true$
$differ(\epsilon, \epsilon' + \rho) = differ(\epsilon, \rho)$ **where** ϵ **and** ϵ' **have no conflict**
$differ(\epsilon, \epsilon' + \rho) = false$ **where** ϵ **and** ϵ' **have conflicts**

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

Figure 5.11 Miscellaneous helper functions.

ure 5.12) and *buildconf* create task configurations for handlers. *buildconf* binds the context variables (of the event type) with the values (ν), computes an unique id for each handler task, and configures the dependent set of this handler (discussed in 3.3). These task configurations are used to run the handler bodies and they are appended to the end of the queue ψ. 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

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

$buildconf(loc, m, \psi, \nu, H, \gamma, \mu) \quad =$
     $\textbf{let } e' = [\texttt{this}/loc, var_1/v_1, \ldots, var_n/v_n]e \quad \textbf{in } \langle e', \langle id, I\rangle\rangle$
 $\textbf{where } loc = [c.F] \textbf{ and}$
 $(c', t, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}, \ldots) = findMeth(c, m)$
 $\textbf{and } \nu = v_1 + \ldots + v_n \textbf{ and } I = pre(loc, m, H, id' + 1, \gamma, \mu)$
 $\textbf{and } id = 1 + car(H) + id' \textbf{and } id' = max(\psi, -1)$

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

$update(\bullet, \gamma, \mu) = \bullet$
$update(\langle \texttt{read}\ \ c\ f\rangle + \rho, \gamma, \mu) = concat(\langle \texttt{read}\ \ c\ f\rangle, update(\rho, \gamma, \mu))$
$update(\langle \texttt{write}\ \ c\ f\rangle + \rho, \gamma, \mu) = concat(\langle \texttt{write}\ \ c\ f\rangle, update(\rho, \gamma, \mu))$
$update(\langle \texttt{create}\ \rangle + \rho, \gamma, \mu) = concat(\langle \texttt{create}\ \rangle, update(\rho, \gamma, \mu))$
$update(\langle \texttt{reg}\ \rangle + \rho, \gamma, \mu) = concat(\langle \texttt{reg}\ \rangle, update(\rho, \gamma, \mu))$
$update(\langle \texttt{ann}\ \rangle + \rho, \gamma, \mu) = concat($
                    $getE(hfind(\gamma, p, \mu), \gamma, \mu), update(\rho, \gamma, \mu))$
```

Figure 5.12 Functions for building handler configurations.

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 γ to union their effect sets with the effect set of this task. Pāñini does this to get more accurate information about the potential effect sets of a task to reduce false conflicts. The functions *diff* and *differ* are used to actually compared the 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.

CHAPTER 6. Properties of Pāñini's Design

In this chapter, we study the key properties of Pāñini's design. We show that our language design has the following desirable properties. Well-typed Pāñini programs do not get stuck and are free of races and deadlocks. The proof of this uses a standard preservation and progress argument [63]. 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 3.3.

6.1 Deadlock Freedom

Definition 6.1.1 *[Blocked Configurations.] A task configuration $\langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle$ may block if any one of its predecessors* [1] *is still in execution.*

Theorem 6.1.2 [Liveness.] *Let $\langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle$ be an arbitrary program configuration, where e is a well-typed expression ,τ is task local data, μ is the store, ψ is a task queue and γ is a handler queue. Then either e is not blocked or there is some task configuration in ψ 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

[1] *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 section 3.3, 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 descripted in Chapter 5).*

initial registration order (Section 3.3) is used to create a non-blocking ordering for the handlers. Finally, leaf nodes, which have no child dependencies, can always either be run concurrently or in an ordering determined by registration time and by 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 Pāñini program does not deadlock.

6.2 Proof of Type Soundness

A standard preservation and progress argument [63] is used to prove the soundness of Pāñini's type system. The details are adapted from previous work [13, 14, 23]. 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 ρ in subsequent discussion. The key definition of consistency is as follows.

Definition 6.2.1 *[Environment-Store Consistency.] Suppose we have a type environment* Π *and* μ *a store. Then* μ is consistent with Π, *written* $\mu \approx \Pi$, *if and only if all the followings hold:*

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

Lemma 6.2.2 [Substitution.] *If* $\Pi, var_1 : t_1, \ldots, var_n : t_n \vdash e : t$ *and* $\forall i \in \{1..n\} \cdot \Pi \vdash e_i : s_i$ *where* $s_i <: t_i$ *then* $\Pi \vdash [var_1/e_1, \ldots, var_n/e_n]e : s$ *for some* $s <: t$.

Proof Sketch: Let $\Pi' = \Pi, var_1 : t_1, \ldots, var_n : t_n$ and $[var'/e'] = [var_1/e_1, \ldots, var_n/e_n]$. 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-Null) and (T-Var), which have no variables and $s = t$. Other cases can be proved by adaptations of MiniMAO$_0$ [13]. 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` $p\ (e_1, \ldots, e_n); e_{n+1}$, 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}, $[var'/e']e_{n+1}$ 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$_0$ [13]. We omit them here.

Lemma 6.2.3 [Environment Extension.] *If* $\Pi \vdash e : t$ *and* $a \notin dom(\Pi)$*, then* $\Pi, a : t' \vdash e : t$.

Lemma 6.2.4 [Environment Contraction.] *If* $\Pi, a : t' \vdash e : t$ *and* a *is not free in* e*, then* $\Pi \vdash e : t$.

Lemma 6.2.5 [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 6.2.6 [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 6.2.7 [Progress.] *For a well-typed expression* e*, a task local data* τ*, a task queue* ψ*, a store* μ*, and a handler queue* γ*. If* $\Pi \vdash e : t$ *and* $\mu \approx \Pi$*, then either*

- $e = loc$ *or*

- $e = \texttt{null}$ *or* $e = \texttt{NullPointerException}$ *or* $e = \texttt{ClassCastException}$ *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 $e = \texttt{null}$, it is trivial.

(b) Cases $e = \texttt{NullPointerException}$ or $e = \texttt{ClassCastException}$ result from the semantics rules $\texttt{null}.f$, $\texttt{null}.f = v$, $\texttt{null}.m(v_1, \ldots, v_n)$, $\texttt{register}(\texttt{null})$ and *cast* e (shown in Figure 5.2). 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 $e = \mathbb{E}[\texttt{new}\ c()]$, $e = \mathbb{E}[loc.m(v_1, \ldots, v_n)]$, $e = \mathbb{E}[loc.f]$, $e = \mathbb{E}[loc.f = v]$, $e = \mathbb{E}[\texttt{cast}\ t\ loc]$, $e = \mathbb{E}[t\ var = v; e]$ and $e = \mathbb{E}[v; e_1]$ are similar to Clifton's work [13, 14] and are omitted.

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

Case $e = \mathbb{E}[\texttt{announce}\ \ p\ (v_1, \ldots, v_n);\ \{e\}]$. 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 $e = \mathbb{E}[\texttt{yield}\ e]$. This case has no constraint and evolves based on different rules. ∎

Theorem 6.2.8 [Subject-reduction.] *Let e be an expression and $e \neq \texttt{yield}\ e_1$ for any e_1, τ task local, ψ a task queue, μ a store, and γ 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$_0$ [13, Section 3.1.4].

The rule for (Sequence) is similar to Clifton's work, except that $e' = \mathbb{E}[\texttt{yield}\ e]$ instead of $e' = \mathbb{E}[e]$. Since the type of $\texttt{yield}\ e$ has the same type as e, this case holds.

For (Define), $e = \mathbb{E}[t\ var = v; e_1]$ and $e' = \mathbb{E}[[var/v]e_1]$: let $\tau' = \tau$, $\mu' = \mu$, $\psi' = \psi$, $\gamma' = \gamma$ and $\Pi' = \Pi$. We now show that $\Pi \vdash e' : t'$ for some $t' <: t$. $\Pi \vdash e : t$ implies that $t\ var = v; e_1$ and all its subterms are well typed in Π. Let $\Pi \vdash (t\ var = v; e_1) : u$. By (T-Define), $\Pi, var : t \vdash e_1 : u'$. By Lemma 6.2.2, $\Pi \vdash [var/v]e_1 : u''$ for some $u'' <: u' <: u$. Therefore, by lemma 6.2.6, $\Pi \vdash e' : t'$ for some $t' <: t$.

For the (Multi-Register) rule, $e = \mathbb{E}[\texttt{register}(v)]$ and $e' = \mathbb{E}[v]$. Let $\tau' = \tau$, $\mu' = \mu$, $\psi' = \psi$, $\gamma' = \gamma$ and $\Pi' = \Pi$. Obviously, $t' = t$.

For the (Register) rule, $e = \mathbb{E}[\texttt{register}(v)]$ 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
$e = \mathbb{E}[\texttt{announce}\ \ p\ \{v_1, \ldots, v_n\}\texttt{;}e_2]$. Let $\mu' = \mu$, $\gamma' = \gamma$, $\Pi' = \Pi$ and $t' = t$. Thus $\Pi \vdash e_2 : t$, has the same type as $\Pi \vdash \texttt{yield}\ e_2 : t$. ∎

Definition 6.2.9 *[Thread-interleaving.] If*
$\langle\langle\mathbb{E}[\textit{yield}\ e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle\mathbb{E}_1[e_1], \tau_1\rangle + \psi_1, \mu_1, \gamma_1\rangle\ \ldots$
$\hookrightarrow \langle\langle\mathbb{E}_n[e_n], \tau_n\rangle + \psi_n, \mu_n, \gamma_n\rangle \hookrightarrow \langle\langle\mathbb{E}[e], \tau\rangle + \psi', \mu', \gamma'\rangle$ *or*
$\langle\langle\mathbb{E}[\textit{yield}\ e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle\mathbb{E}[e], \tau\rangle + \psi', \mu', \gamma'\rangle$,
we denote this as $\langle\langle\mathbb{E}[\textit{yield}\ e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow^*$
$\langle\langle\mathbb{E}[e], \tau\rangle + \psi', \mu', \gamma'\rangle$, *where* $\forall i\{1 \leq i \leq n\}\ \mathbb{E}_i[e_i] \neq \mathbb{E}[e]$.

Theorem 6.2.10 [Subject-reduction-Thread-interleaving.] *For an expression* $e = \mathbb{E}[\textit{yield}\ e_1]$ *for any* e_1, τ *task local data, and* ψ *a task queue,* μ *a store and* γ *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 any data races (because when an event is announced, Pāñini schedules tasks to eliminate races and to maximize concurrency, as discussed in 3.3) and thus, 1 of Definition 6.2.1 holds. Since the store μ does not shrink, 2 of Definition 6.2.1 holds. Clearly, `yield` e_1 and e_1 have the same type in Π, and therefore, by Lemma 6.2.5, if $\Pi \vdash e : t$, then $\Pi \vdash e' : t'$. ∎

Theorem 6.2.11 [Soundness.] *Given a program $P = \mathtt{decl}_1 \ldots \mathtt{decl}_n\ e$, if $\vdash P\ : (t, \rho)$ for some t and ρ, then either the evaluation of e diverges or else $\langle\langle e, \langle 0, \emptyset\rangle\rangle, \bullet, \bullet\rangle \hookrightarrow^* \langle\langle e', \tau'\rangle + \psi', \mu', \gamma'\rangle$ where one of the following holds for v:*

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

Proof Sketch: If e diverges, then this case is trivial. Otherwise if e converges, it is true because the empty environment is consistent with the empty store. This case is proved by Theorem 6.2.7, Theorem 6.2.8 and Theorem 6.2.10. ∎

CHAPTER 7. Pāñini's Compiler and Runtime System

To a certain extent, implementing Pāñini as a library is feasible [48]. However, to get deadlock and race freedom and a deterministic semantics, which is crucial for writing correct and efficient concurrent programs, programmers will need to write extensive effect annotations (like Jade [53]). This could be tedious and error prone so we implemented a compiler for Pāñini using the JastAddJ extensible compiler system [18]. This compiler and associated examples are available for download from `http://paninij.org`.

As its backend, Pāñini's runtime system uses the fork/join framework [34], which is a fast lightweight task framework built upon Java threads, and geared for parallel computation. This framework uses the work stealing algorithm [8] and works well for recursive algorithms. We observed that handlers usually also act as subjects and recursively announce events, thus Pāñini 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 framework and may execute concurrently. Below we describe key parts of our implementation strategy.

Event type. An event type declaration is transformed into an interface (an example is shown in Figure 7.1). A getter method is generated for each context variable of the event (`Generation g()` on line 2 in Figure 7.1) so that the handlers can use this method to access the context variables. Two interfaces, namely `EventHandler` (lines 3–5) and `EventPublisher` (line 6), are to be used by an inner class `EventFrame` (lines 7–20), 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.

```
1   public interface GenAvailable {
2     public Generation g(); //An accessor for each context variable
3     public interface EventHandler extends IEventHandler{
4       public void GenAvailableHandle(Generation g);
5       public AbstractReferenceSet genAvailableSet();
6     }
7     public interface EventPublisher extends IEventPublisher{ }
8     public class EventFrame implements GenAvailable {
9       public static void register(IEventHandler handler) {
10        //1. check whether this handler has registered before,
11        //   if yes return ( no duplicate registration )
12        //2. analyze the effects of the handler
13        //3. insert it into the handler hierarchy
14      }
15      public static void announce(GenAvailable ev) { /* explain later */ }
16    } //other helper methods elided
17    public static class GenAvailableTask extends PaniniTask { .. }
18  }
```

Figure 7.1 An event type is translated into an interface. Snippets from translation of event `GenAvailable` in Figure 1.2.

Event Announcement. When a subject signals an event, the announce method (line 14 in Figure 7.1) is called (Figure 7.2). This method iterates over the handlers and executes all non-conflicting handlers as discussed in Section 3.3. The class `EventFrame` uses a helper class (here `GenAvailableTask` on line 21), to wrap the handlers (if any) before submitting them for execution.

```
public static void announce(GenAvailable ev) {
  /* iterate each level in the hierarchy */
  for(int i = 0; i < levels; i++ ){ // number of levels in the hierarchy
    GenAvailableTask[] ct;
    final int tempSize = num_level[i]; // number of handlers in each level
    ct = new GenAvailableTask[tempSize];
    // for later used by the Fork/Join framework
    /* elements: a two dimensional array holding all the handlers */
    EventHandler[] ehs = elements[i];
    /* wrap each handler, in each level, into a PaniniTask and
       put it to the Fork/Join framework later */
    for(int j = 0; j < tempSize; j++ ){
      ct[j] = new GenAvailableTask(ehs[j], ev);
    }
    PaniniTask.coInvoke(ct);  // handlers in the same level run concurrently
  }
}
```

Figure 7.2 Full code of the announce methods

Handler Registration. A register method is added to every class that has event bindings (Figure 7.3). For example, in Figure 7.4, the method `_panini_register` (on lines 5-7) is added to the class `CrossOver`, since it has a corresponding binding as shown in Chapter 1.

This method in `CrossOver` will call the `register` method (Figure 7.1, line 8) in the event frame.

```
public static void register(IEventHandler handler) {
  if( elements == null ){ // the first handler for this event
       initialize the bookkeeping fields of this event
       put the handler in the first level and return
  }
  /* more than one handlers */
  for ( i in each level ){
    if ( handler == elements[i][j] ) // for each element in level i
      return;      //if the handler is already in the list, do nothing
  }
  // get the effect of the handler
  AbstractReferenceSet otherEffect = handler.genAvailableSet();
  if( otherEffect is not a subset of the current effect of the event ){
    enlarge the effect of all the publishers for this event
  }
  /* iterate each level reversely in the hierarchy to find
     a level that has handler conflict with the subscriber */
  for ( i in each level ){
    if( otherEffect conflicts with effect of elements[i][j] ){
      put handler in level i+1 and return
    }
  }
}
```

Figure 7.3 Pseudo code of the register methods

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 (lines 8–13 in Figure 7.1). This method will first check whether the current registering handler is already in the handler hierarchy to ensure no duplicate registration. Then the effects of the newly registered handler are compared against other previously registered handlers to calculate the dependence set of this handler (as discussed in Section 3.3). Finally, the handler is put into a proper level in the hierarchy.

Event Handler. Every handler is transformed as follow. An effect computing method, which compute the effect of the original method, is inserted into the handler class. For example, on lines 11-14, effect computing method `_panini_effect_cross_LGeneration` is generated and will compute the effect representation for the original method `cross`. The getter method (on lines 16-18) for the effect representation is used by the corresponding `register` method in the event frame; in this example, it is used by the method in Figure 7.1 on lines 8-13.

```
class CrossOver {
    /* unrelated fields, constructors and methods,
       which are the same as shown in previous sections,
       are elided */
    void _panini_register(EventHandler subscriber){
        GenAvailable.EventFrame.register(subscriber);
    }
    public void cross(Generation g){
        /* method body is elided*/ }
    AbstractReferenceSet ars_GenAvailable = new AbstractReferenceSet();
    AbstractReferenceSet _panini_effect_cross_LGeneration(){
        /* construct read, write, announce and register effects as
        discussed in previous chapters */
        return ars_GenAvailable;
    }
    public AbstractReferenceSet effectset_GenAvailable() {
        return ars_GenAvailable;
    }
}
```

Figure 7.4 A handler is translated to extend the EventHandler interface. Snippets from translation of `CrossOver` in Figure 1.2.

CHAPTER 8. Evaluation

We now evaluate the design and performance benefits of Pāñini. All experiments were run on a system with a total of 12 cores (two 6-core AMD Opteron 2431 chips) running Fedora GNU/Linux.

8.1 Analysis of Modularity and Concurrency Synergy

Our goal is to analyze "if a program is modularized using Pāñini does that also expose potential concurrency in its execution?"

We have already presented one such case in Chapter 1, where modularization of various concerns in the implementation of a genetic algorithm exposed potential concurrency between these concerns. We now analyze the speedup of the genetic algorithm implementations presented in Figure 1.1 and Figure 1.2. Recall that the first version is implemented by taking the sequential version and retrofitting it with thread and synchronization primitives, whereas the second version is implemented by modularizing the code. We first compared these implementations head-to-head. The results for this comparison are shown as black bars in Figure 8.1.

In this experiment, the average speedup over ten runs was taken with a generation (or population) size of 3000 and a depth (number of generations) of 10. For a variety of generation sizes (1000–3000) and depths (8–11), speedups were similar.

The results show that Pāñini's implementation achieved between 1 and 4x speedup for varying number of threads. This was quite surprising as we expected the concurrent version in Figure 1.1 to match or exceed the performance of Pāñini's version since the OO version does not incur the overhead of implicit concurrency.

A careful analysis by a seasoned concurrent programmer revealed two problems with this

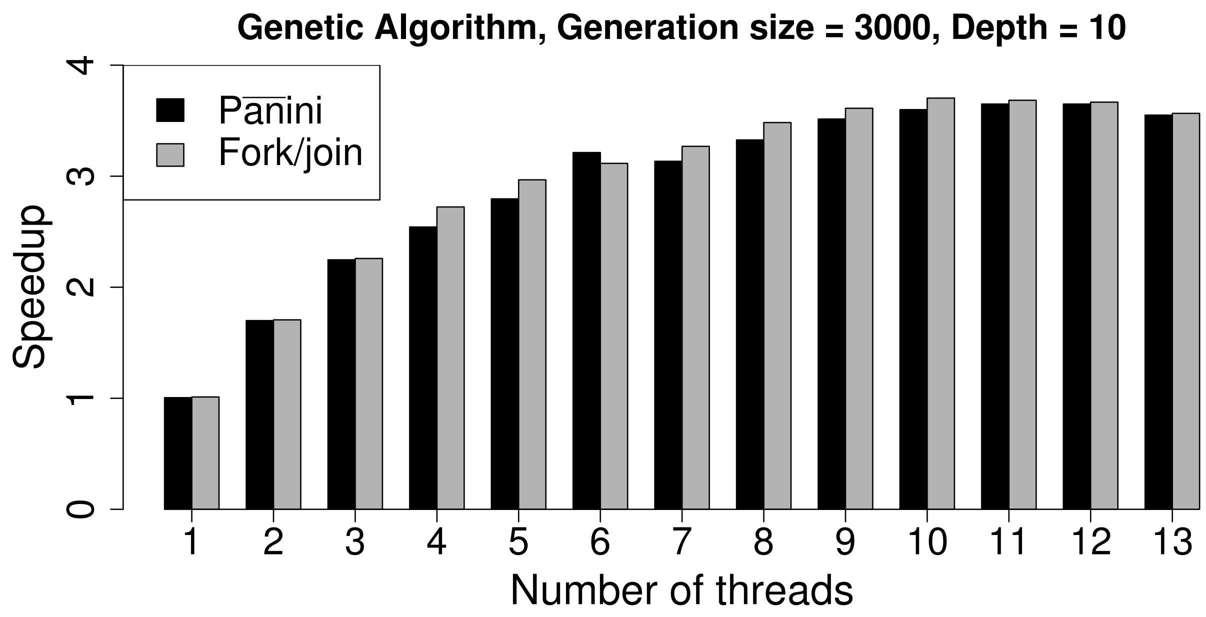

Figure 8.1 Speedup over sequential OO code (black bar: Pāñini; Gray bars: hand tune fork/join code).

seemingly straightforward concurrent code in Figure 1.1. Our expert pointed that: "*the entire genetic algorithm code is wrapped in a future task. The method then submits the future task on line 30 and immediately invokes the method* `get`*, which limits concurrency. Furthermore, the compute() method calls (on line 16 and 23) are synchronous method calls, and thus, the two subtasks could not be run concurrently. As a result, the algorithm execution proceeds as a depth-first search tree (the right subtree will not be executed until the left subtree is done) but the intention is to execute the branches of the search tree concurrently.*"

This analysis was both shocking and pleasant. Shocking in the sense that even with a relatively simple piece of concurrent code, correctness and efficiency were hard to get. Pleasant in the sense that the Pāñini code automatically dealt with these problems.

Following our concurrency expert's advice, we created a second version of the object-oriented genetic algorithm using the fork/join framework [34]. The performance results of this "expert version" is shown in Figure 8.1 as gray bars. This figure shows that the speedups between the "expert version" and the Pāñini versions for this genetic algorithm are comparable.

In summary, our performance evaluation revealed correctness and efficiency problems with a relatively straightforward OO parallelization of the genetic algorithm, whereas Pāñini's implementation didn't have these problems. Fixing the problems with OO implementation by an expert led to comparable performance between implicit concurrency exposed by Pāñini and explicitly tuned concurrency exposed using the fork/join framework [34].

8.2 Performance Evaluation

The goal of this section is to analyze "how well do the Pāñini programs perform compared to a hand-tuned concurrent implementation of equivalent functionality?" We first describe our experimental setup and then analyze speedup realized by Pāñini's implementation as well as the overheads.

8.2.1 Concurrency Benchmark Selection

To avoid bias and subtle concurrency problems similar to Section 8.1, 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 [34].

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 [34],
2. a Pāñini version with no conflict between handlers, and
3. a second Pāñini's implementation that was intentionally designed to have conflicts between handlers.

To introduce conflicts, we add another handler that aggregates the results of concurrently executing handlers. Thus, the third handler must wait for the other handlers to complete since it depends on them. 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 task. When both of these tasks are completed, the spawning task adds them together. For Pāñini, 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.

8.2.2 Speedup over Sequential Implementation

Figure 8.2 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.

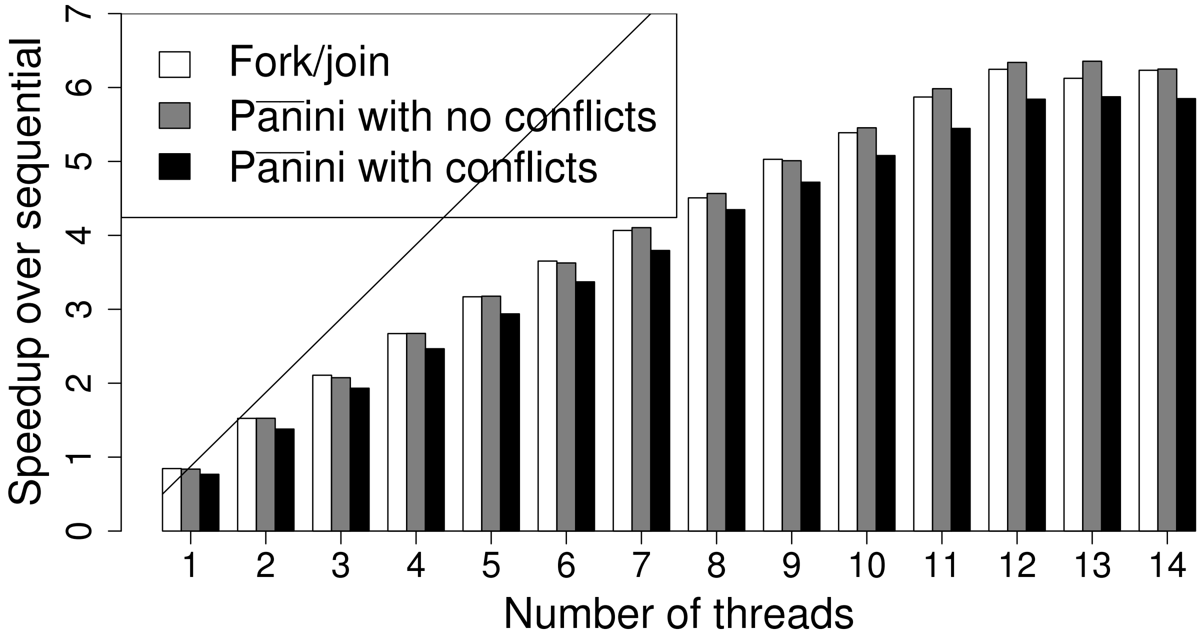

Figure 8.2 Average speedup over sequential version (Line: perfect scaling).

This figure shows that the speedups between the three styles are comparable. Speedups for fork/join and Pāñini without conflicts are nearly the same. A statistical analysis showed that for all benchmarks, we do not see a statistically significant difference ($p < 0.05$) between fork/join and Pāñini with no conflicts.

From the figure, we can also see that Pāñini with conflicts has slightly lower speedup than both fork/join and Pāñini without conflicts, however, this decrease is rather small (average 6.5% decrease from fork/join).Note that since we are using a machine with 12 cores, performance levels drop off at 12 threads.

8.2.3 Overhead over the Sequential Implementation

We also measured the overhead involved with Pāñini as compared to the standard fork/join model. We first consider the average overhead across all benchmarks as shown in Figure 8.3. Overhead is computed by determining the increase in runtime from the sequential version to the *concurrent version with a single thread.* For this experiment, we used large input sizes.

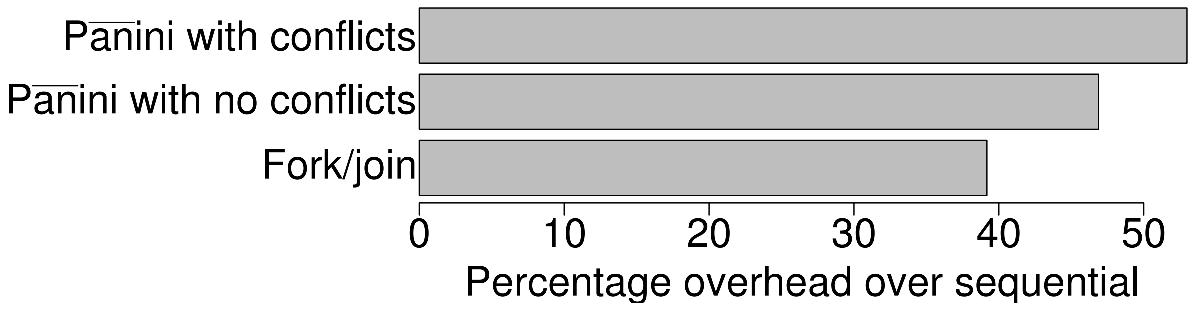

Figure 8.3 Average overhead over sequential version for each technique.

This figure shows us that while Pāñini increases the overhead over fork/join, it is not a prohibitive amount. For example, for Pāñini with no conflicts, we only see a 7.7% increase in overhead.

Figure 8.4 shows a summary comparison of overhead as program input size changes. In this figure, the overhead for 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. Here, overhead decreases

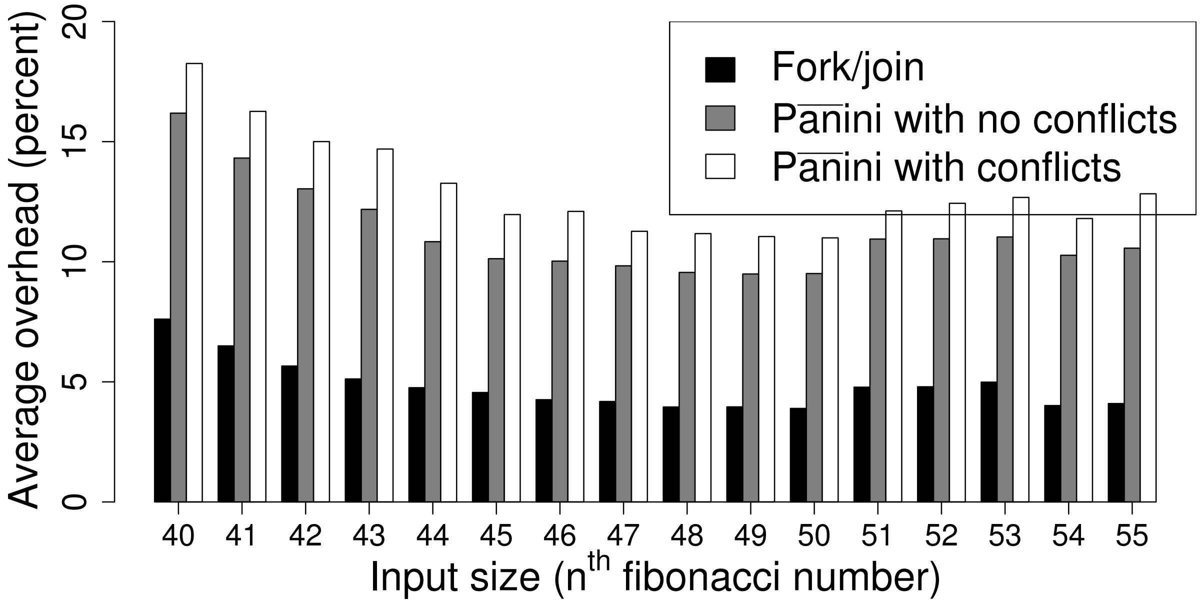

Figure 8.4 Average overhead for Fibonacci benchmark for varying input size and each scheduling strategy.

to as low as 5.5% additional overhead for Pāñini with no conflicts. Pāñini with conflicts only incurs an additional 1.2% overhead for larger input sizes. Each of the differences in overhead (fork/join vs. Pāñini without conflicts, fork/join vs. Pāñini with conflicts, and Pāñini with vs. Pāñini without conflicts) was always statistically significant ($p < 0.05$).

8.3 Summary of Results

In summary, Pāñini shows speedups which scale as well as expert code in the standard fork/join model. Even though Pāñini has a higher overhead than fork/join, Pāñini 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 deterministic semantics.

8.4 Other Examples in Pāñini

To further assess Pāñini'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.

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 from the problem of code-tangling: implementations of different concerns (i.e., transformation tasks) are all mixed together.

```
class MethodDecl extends ASTNode {
  Expression body; // the expression body of the method
  /* other fields and method elided */
  Effect computeEffect(){
    return body.computeEffect(); }
}
class Expression extends ASTNode{
  Effect computeEffect(  );
class Sequence extends Expression{
  Expression left; Expression right;
  Effect computeEffect(){
    Effect effect = left.computeEffect();
    effect.add( right.getEffect() );
    return effect; }
}
class FieldGet extends Expression{
  Expression left; /* other fields elided */
  Effect computeEffect(){
    Effect effect = left.computeEffect();
    effect.add( new ReadField() );
    return effect;}
}
```

Figure 8.5 Snippets of an AST with an Effects System

Figure 8.5 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 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 [18]. The visitor design pattern solves this problem to a certain extent but it has other problems [18].

```
 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 8.6 Pāñini's version of visiting an abstract syntax tree.

Pāñini handles this modularization problem readily as shown in Figure 8.6. 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 the effect concern is modularized as the class `ComputeEffect`. This class has two bindings 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 Pāñini program manifests a few design advantages. First, the AST implementation is completely separated from effect analysis. Also, unlike the visitor pattern, the `ComputeEffect` class need not implement default functionality for all AST nodes. Furthermore, other passes such as type checking, error reporting and code generation etc., can also reuse the AST events.

Last but not least, in Pāñini, 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, Pāñini's type system will detect this as interference and automatically generate a schedule of their execution that would be equivalent to sequential execution. Thus, for this example Pāñini shows that it can reconcile the modularity and concurrency goals such that modular design of compilers also improves their performance on multi-core processors.

Modular and Concurrent Image Processing. This example is adapted from and inspired by the ImageJ image processing toolkit [29]. 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 24–29) maintains a list of pixels. The method `set` for this class (lines 27–29) 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-Base Thresholding [25]. Thresholding is a method for image segmentation that is typically used to identify objects in an image. 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 from the image class to maximize reuse of the image class. Figure 8.7 shows the implementation of two thresholding

```
event Changed{  Image pic; }
class Percentile {
  Hashtable h; int p /* Percentile value */
  Percentile(int percentile){
    register(this); h = new Hashtable(); this.p = percentile;
  }
  when Changed do compute;
  void compute(Image pic){
  /* threshold is the intensity value for which cumulative
    sum of pixel intensities is closest to the percentile p.*/
  h.add(pic, threshold);
}}
class GlasbeyThreshold {
  Hashtable h;
  GlasbeyThreshold (){
    register(this); h = new Hashtable();
  }
  when Changed do compute;
  void compute(Image pic){
  /* threshold is the intensity value for which cumulative
   sum of pixel intensities has the most dominant value. */
  values.put(pic, threshold);
}}
class Image {
  List pixels;
  Image set(Integer i, Integer v){
    pixels.setAt(i,v);
    announce Changed(this);
}}
```

Figure 8.7 An Image and Threshold Computation in Pāñini.

methods in classes `Percentile` and `GlasbeyThreshold`. Pāñini's implementation allows the threshold computation concerns to remain independent of the image concerns, while allowing their concurrent execution.

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. For example, class `CheckAirline` extracts source and destination information from the trip data and stores the flight results by calling the method `setFlight`. Similarly, the class `CheckFlight` computes and stores the hotel results and `CheckRentalCar` computes and stores the car rental search

```
event PlanTrip{ TripData d; }  //Event Type
class CheckAirline { //Searches for available flights.
  List<Airline> alist;
  CheckAirline(List<Airline> l){register(this); this.alist = l;}
  when PlanTrip do checkFlights;
  //Find all the available flights during the trip
  void checkFlights(TripData d){
    for(Airline a : alist) {
      Flight flight = a.getFlights(d.from(),d.to());
      //add the results to the tripData
      d.setFlight(flight);
}}}
class CheckHotel { //Searches for available hotels.
  List<Provider> hlist;
  CheckHotel(List<Provider> l){register(this); this.hlist = l;}
  when PlanTrip do checkHotels;
  void checkHotels(TripData d){
  for(Provider h: hlist) {
    Hotels hotels = h.search(d.from(),d.to(),d.pricePref());
    d.setHotels(hotels);
}}}
class CheckRentalCar { //Searches for available cars.
  List<Agency> clist;
  CheckRentalCar(List<Agency> l){register(this); this.clist = l;}
  when PlanTrip do checkCarRentals;
  void checkCarRentals(TripData d){
  for(Agency c: clist){
    Cars cars = c.getRentals(d.from(),d.to(),d.carPref());
    d.addRentalChoices(cars);
}}}
```

Figure 8.8 Accessing service providers in handlers.

results. In this example as well Pāñini's design shows the potential of reconciling modularity goals with concurrency goals. When an event of type `PlanTrip` is announced, each of the three handler methods can execute concurrently.

Concurrent Refactoring Detection. Our next example presents tool for detecting refactoring between softwares [17]. Currently, the tool detects refactorings like renaming and changes of method signatures. These refactoring detection algorithms could be run independently.

As we could see in Figure 8.9, different refactoring detection algorithms, namely rename package, rename class, rename method, pullup method and push down method detection, are scattered and seriously tangled with concurrency concerns in the original implementation. Also, it may not be easy to dynamically introduce new detection algorithms.

Pāñini handles this modularization problem readily as shown in Figure 8.10. In this example, different handlers are no longer tangled with each other, and more other handlers could

```
detectRefactoring(){
  endGate = new CountDownLatch(4);
  renamePackageThread = new Thread(){
    run(){
      try{
        detectRenamePackage(utility, v1Graph, v2Graph);
      } finally{
          endGate.countDown();
      }}};
  renamePackageThread.start();
  renameClassThread = new Thread(){
    run (){/*detail omitted*/};
  renameClassThread.start();
  renameMethodThread = new Thread(){
    run (){/*detail omitted*/};
  renameMethodThread.start();
  pullUpThread = new Thread(){
    run (){/*detail omitted*/};
  pullUpThread.start();
  pushDownThread = new Thread(){
    run (){/*detail omitted*/};
  pushDownThread.start();
  endGate.wait();
}
```

Figure 8.9 Refactoring Detection with Java concurrency utilities.

be readily introduced.

When an event of type `Detection` is announced, these five strategies could execute concurrently. Each strategy checks appropriate pairs of entities and has access to the graphs v1Graph and v2Graph, and the utility as input from the users. RenamePackage, RenameClass, and RenameMethod strategies are quite similar. They do a few checks to eliminate false positives. PullUpMethod and PushDownMethod are the opposite of each other. PullUpMethod pulls up the declaration of a method from a subclass into the superclass such that it could be reused by other subclasses. PushDownMethod pushes down the declaration of a method from a superclass into a subclass that uses the method because this method is no not used by other subclasses any more.

Figure 8.11 shows a summary speedup of the Pāṉini version over the sequential versions. In this figure, the average speedup was taken. For each program, Eclipse projects version 2.1.3 and version 3.0 were used as input.

Performance Results. Modularization of the effects analysis and image analysis resulted in speedup of roughly 2x. The modularization of service requests gave speedups around

```
event Detection{ String v1Graph; String v2Graph; int utility; }  //Event Type
class renamePackageHandler{
  //detail elided
  when Detection do detect;
  void detect( String v1Graph, String v2Graph, int utility){
    detectRenamePackage(utility, v1Graph, v2Graph); }
}
class renameClassHandler{
  //detail elided
  when Detection do detect;
  void detect( String v1Graph, String v2Graph, int utility){
    detectRenameClass(utility, v1Graph, v2Graph); }
}
class renameMethodHandler{
  //detail elided
  when Detection do detect;
  void detect( String v1Graph, String v2Graph, int utility){
    detectRenameMethod(utility, v1Graph, v2Graph); }
}
class pullUpMethodHandler{
  //detail elided
  when Detection do detect;
  void detect( String v1Graph, String v2Graph, int utility){
    detectPullUpMethod(utility, v1Graph, v2Graph); }
}
class pushDownMethodHandler{
  //detail elided
  when Detection do detect;
  void detect( String v1Graph, String v2Graph, int utility){
    detectPushDownMethod(utility, v1Graph, v2Graph); }
}
```

Figure 8.10 Pāṇini's version of Refactoring Detection.

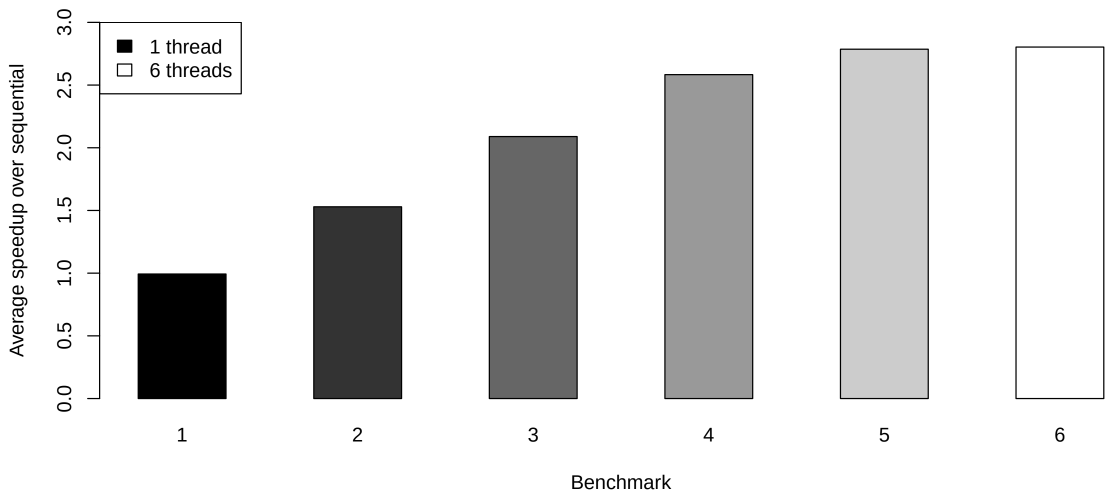

Figure 8.11 Refactoring Crawler: Average speedup over sequential version.

3x. For refactoring detection, we observed speedups around 3x. These values were as expected, based on the available concurrency in the problems. Moreover, this scalability is obtained without requiring programmers to write a single line of explicitly concurrent code.

CHAPTER 9. Conclusion and Future Work

9.1 Conclusion

Language features that promote concurrency in program design have become important [6]. Explicit concurrency features such as threads are hard to reason about and building correct software systems in their presence is difficult [44]. 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 Pāñini, we pursue this goal. In an effort to do so, we have developed the notion of asynchronous, typed events that are especially helpful for programs where modules are decoupled using implicit-invocation design style [42, 16, 58]. Event announcements provide implicit concurrency in program designs when events are signaled and consumed. We have tried out several examples, where Pāñini improves both program design and potential available concurrency. Unlike message-passing languages such as Erlang [5] the communication between implicitly concurrent handlers is not limited to value types or record of value types.

An important property of Pāñini's design is that, for systems utilizing implicit-invocation design style, it makes scalability a by-product of modularity. For example, observe that in genetic algorithm, AST analysis, image analysis, and trip planning addition of new modules in a non-conflicting manner doesn't affect the scalability of existing modules. For example, a new observer for PlanTrip event (say sightseeing) would run concurrently with other observers. Similarly, a new thresholding observer could also run concurrently with other observers for `Changed` event.

9.2 Future Work

Future work includes extending Pāñini's design, semantics and implementation in several dimensions.

Library Based Approach. Pāñini requires changing the Java compiler in the current implementation, which is not convenient in some sense. In the future, we are planning to offer programmers with a set of libraries, such that, no changes to the compiler are necessary. In the same time, these libraries have to ensure the safety of accessing the shared heap. Because of this, Pāñini has to also present a set of libraries such that programmers could adhere to certain protocol and eliminates concurrency errors.

Effect Set Detection. We have presented a conservative mechanism (effect sets for handlers [27, 36]) for detecting conflicts between handlers, so it would be good to study and improve its precision. Further analysis such as dynamic point to analysis may enhance the precision of detecting false interference. On the other hand, the idea of region [11, 12] could be applied and Pāñini could provide a better analysis of whether the handlers are accessing disjoint regions.

Software Design Patterns. Also, we are planning to use the effect set detection scheme presented in this work and would like further investigate whether it could be applied to other software patterns [24]. For example, chains of responsibility [24, pp 223] and iterator patterns [24, pp 257] have the same action commands for a certain period without changing. Therefore, the dynamic effect set detection, which occupy a small portion of the lifetime of the program could be amortized by concurrently applying the commands to the context.

Asynchronous Event Announcement. In the current implementation, handlers that do not have conflict between each others could be run in concurrent. However, publishers have to block and wait for the handlers. In the next step, we would like to study patterns that could enable asynchronous announcement, in which the base code could advance to certain point. Because of this, more concurrency could be exposed. Then, we have to change the implementation to take into account, the effects of the continuation of the publishers.

BIBLIOGRAPHY

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[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] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multi-cores with aspect-oriented programming. In *AOSD*, pages 1–12, 2010.

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

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

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *the conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 81–96, 2009.

[8] 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.

[9] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on*

Object-oriented programming, systems, languages, and applications, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[10] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM.

[11] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 441–460, New York, NY, USA, 2007. ACM.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 292–310, New York, NY, USA, 2002. ACM.

[13] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.

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

[15] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Conference on Hot Topics in Operating Systems*, Berkeley, CA, USA, 2005.

[16] David C. Luckham *et al.*. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, 1995.

[17] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[18] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA*, pages 1–18, 2007.

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

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

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

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

[23] 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.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[25] C. A. Glasbey. An analysis of histogram-based thresholding algorithms. *CVGIP: Graphical Models and Image Processing*, 55(6):532 – 537, 1993.

[26] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[27] A. Greenhouse and J. Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

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

[29] Image Processing and Analysis in Java. ImageJ. `http://rsbweb.nih.gov/ij/`.

[30] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 31:1:1–1:48, December 2008.

[31] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX*, 2007.

[32] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[33] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

[35] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *the conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 227–242, 2009.

[36] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 246–257, New York, NY, USA, 2002. ACM.

[37] 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.

[38] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 359–371, New York, NY, USA, 2006. ACM.

[39] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[40] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[41] Mohsen Vakilian and Danny Dig and Robert Bocchino and Jeffrey Overbey and Vikram Adve and Ralph Johnson . Inferring method effect summaries for nested heap regions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 421–432, 2009.

[42] 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.

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

[44] J. Ousterhout. Why threads are a bad idea (for most purposes). In *ATEC*, January 1996.

[45] P. Charles *et al.*. X10: an object-oriented approach to non-uniform cluster computing. In *the conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 519–538, 2005.

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

[47] R. Bocchino *et al.*. A type and effect system for deterministic parallel java. In *the conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 97–116, 2009.

[48] H. Rajan, S. M. Kautz, and W. Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *2010 Onward! Conference*, October 2010.

[49] 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.

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

[51] 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.

[52] 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.

[53] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[54] J. Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

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

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

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

[58] K. J. Sullivan and D. Notkin. Reconciling Environment Integration and Component Independence. *SIGSOFT Software Engineering Notes*, 15(6):22–33, December 1990.

[59] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[60] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

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

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

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

[64] A. Yonezawa. ABCL: An object-oriented concurrent system, 1990.

[65] A. Yonezawa and M. Tokoro. Object-oriented concurrent programming, 1990.