12-15-2010

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

Yuheng Long
Iowa State University, csgzlong@iastate.edu

Hridesh Rajan
Iowa State University

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

Yuheng Long and Hridesh Rajan

Initial Submission: December 15, 2010.

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

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

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

Yuheng Long and Hridesh Rajan

Iowa State University, Ames, Iowa, USA
{csgzlong,hridesh}@iastate.edu

Abstract. Driven by the need to utilize multicore platforms, recent language designs aim to bring the concurrency advantages of events in *distributed publish-subscribe* systems to sequential OO programs that utilize the *implicit invocation* (II) design style. These proposals face two challenges. First, unlike the *publish-subscribe* paradigm where publisher and subscriber typically do not share state, communicating via shared state is common in II programs. Second, type-and-effect systems that are generally designed for statically reasoning about a program's execution are often too conservative to handle II that typically entails a virtual method invocation on zero or more dynamically registered handlers. To solve these problems, we present a novel hybrid type-and-effect system for exposing concurrency in programs that use II mechanisms. This type-and-effect system provides deadlock and data race freedom in such usage of II mechanisms. We have also implemented this type-and-effect system. An initial study shows its scalability benefits and acceptable costs.

1 Introduction

The implicit invocation (II) style has seen significant usage in the design of object-oriented (OO) programs, e.g. via the observer design pattern [1], where it helps improve modularity [2]. The basic idea is that some components (subjects) signal events. Other components (handlers) register to listen to these events. When subjects signal events, handlers are invoked implicitly, which helps decouple subjects from handlers [2].

In the distributed systems area, similar ideas have been developed under the umbrella term *publish/subscribe* systems [3–5]. In such systems, events help decouple the execution of components thereby exposing potential concurrency in system design [5].

With emerging multicore platforms, finding concurrency is becoming vital for scalability of shared-memory programs. The research question then is: Can II style shared-memory programs enjoy the concurrency benefits of events, just like publish-subscribe based distributed systems? Existing work has explored this question [6–10]. Even mainstream languages such as $C\sharp$ have added features for exposing concurrency in programs that use implicit invocation design style [11]. Polyphonic $C\sharp$ has a similar feature [7].

1.1 A Running Example

To illustrate the use of implicit invocation towards exposing potential concurrency, consider the example in Figure 1 that implements an e-mail client with spam filtering.

* Rajan and Long were supported in part by the US NSF under grant CCF 08-46059.

```
1 class Client {
2  void receive(Email e){
3   notify(e);
4   ... //Further process email (elided)
5  }
6  List fs=new ArrayList();
7  void addFilter(Filter f){  fs.add(f); }
8  void notify(Email e){
9   for(Filter f: fs) f.filter(e);
10 }
11 }
12 interface Filter{
13   void filter(Email e);
14 }
15 interface Action{
16   void act(Email e, Filter f);
17 }
18 class Logger implements Action{
19  List l = new ArrayList();
20  void act(Email e, Filter f){
21   l.add(e.toString()+f.toString());
22  }
23 }
24 abstract class SpamFilter implements Filter{
25  void init(Client c){ c.addFilter(this); }
26  List as=new ArrayList();
27  void addAction(Action a){ as.add(a); }
28  void notifyActions(Email e, Filter f){
29   for(Action a: as) a.act(e, f);
30  }
31 }
32 class Bayesian extends SpamFilter {
33  void filter(Email e){
34   if(isSpam(e)) notifyActions(e,this);
35  }
36  boolean isSpam(Email e){
37   ... //Uses Bayesian spam detection
38  }
39 }
40 class Markovian extends SpamFilter {
41  void filter(Email e){
42   if(isSpam(e)) notifyActions(e, this);
43  }
44  boolean isSpam(Email e){
45   ... //Uses Markovian spam detection
46 }}
47 /****  Main method   *********/
48  Client c = new Client();
49  Bayesian bf = new Bayesian();
50  Markovian mf = new Markovian();
51  Logger l = new Logger();
52  bf.init(c); mf.init(c);
53  if(...){ bf.addAction(l); mf.addAction(l)}
```

Fig. 1. Snippets from an e-mail application that uses implicit invocation.

For future evolution, e.g. adding/removing spam filters, it is desirable that the class `Client` remains independent of spam filters. This decoupling is achieved using the observer pattern [1] with the class `client` as subject and spam filters as handlers. The class `client` includes a field `fs` to store active filters, provides a method `addFilter` to add a filter (`removeFilter` elided), and a method `notify` to invoke each filter in the list `fs` without naming its concrete class. The spam filters classes implement the interface `Filter` (lines 12-14) that has a handler method `filter`.

Once an e-mail is classified as spam, e-mail clients may want to take certain `Action`(s), e.g. logging it to keep a record of spam. These `Action`s are triggered by the spam filters; however, the `Action` depends on the e-mail client. Thus, it would be sensible to keep the implementation of `Action`s separate from the implementation of spam filters, so that both can be reused. This is also achieved using the observer pattern. In this scenario, spam filters are subjects and `Action`s are handlers.

Similar to the subject `Client`, the class `SpamFilter` implements the functionality to store a list of handlers (`Action` instances), adding to this list, and to run all actions (on lines 24-31). Finally, concrete spam filters `Bayesian` and `Markovian` implement different methods of spam filtering (details are omitted, but are discussed in other works [12, 13]).

The main method (lines 47-53) creates an instance of `Client`, instances of filters `Bayesian` and `Markovian` and calls the method `init` (line 52) to register these filters with the client. The main method also creates an instance of the desired `Action`s, e.g. `Logger` in Figure 1 and may call the method `addAction` (line 53) to register these `Action`s with filters. When an e-mail is received, the registered filters process it. If the e-mail is classified as spam, loggers, if registered, will be invoked.

1.2 The Problems and their Importance

To enhance the scalability of this e-mail client, it may be desirable to expose concurrency in its design. One such opportunity arises in spam filtering. Both `Bayesian` and `Markovian` filters are compute-intensive. Thus processing an e-mail simultaneously using both filters can enhance the scalability and responsiveness of this e-mail client.

```
1 void notify(final Email e){
2  int size = fs.size();
3  Thread[] ts = new Thread[size];
4  for( int i=0; i<size; i++ ){
5   final Filter f = fs[i];
6   ts[i] = new Thread(
7     new Runnable() {
8      void run(){ f.filter(e); }
9    });
10   ts[i].start();
11  }
12   for( int i=0; i<size; i++ ){
13     ts[i].join();
14   }
15 }
```

Fig. 2. Concurrent version of the method `notify` in class `Client` from Figure 1.

Figure 2 shows a common idiom for exposing this concurrency between filters using Java threads. This idiom is effective, but it suffers from two problems: data races and non-deterministic semantics. Existing work has proposed novel type-and-effect systems [14, 15] to solve these problems [16–18]. The basic idea behind a type-and-effect system is to statically check effects such that these checks are a conservative approximation of those that may be performed by a dynamic semantics instrumented with the same effect system [15]. A static type-and-effect system for validating concurrent programs may declare many programs concurrency-unsafe, even though only certain rare control flow paths in such programs produce concurrency-unsafe computational effects.

Programs that require sound guarantees must accept this conservative approximation because, in general, the runtime overhead of instrumentation essential to determine if one such concurrency-unsafe control flow path is about to run is often prohibitive.

The need for conservative approximation in a type-and-effect system for programs can arise due to two features: data structures with variables and dynamically varying number of elements such as a `List`, and dynamic dispatch. The computational effects of an operation on such a data structure must be taken as the upper bound of effects produced by this operation on any possible state of that data structure. The computational effects of a dynamically dispatched method call must be taken as the upper bound of effects produced by all overriding implementations of the called method.

Unfortunately, both of these features are used in a typical observer pattern idiom as seen in Figure 1 on lines 6-10 and 26-30.

To illustrate the effect of this conservative approximation on potential concurrency, let us consider the concurrent implementation of the `notify` method in Figure 2. Informally, the method `notify` is concurrency-safe if the effects of the `filter` methods in each concrete spam filter are disjoint, since any concrete filter can be put into this list by passing it as an argument to the method `addFilter` on line 6 in Figure 1.

The effects of the `filter` method in classes `Bayesian` and `Markovian` would be the same as the effects of the method `notifyActions` (since the method `isSpam` is pure in both these classes). This method could invoke any combination of the `Actions`

(or none of them at all, depending on which `Action`s have registered). Thus, we must include the effects of all the `Action` handlers in the effect set of `notifyActions`, to make it valid for all inputs. The method `act` in the class `Logger` adds an element to the List `l`, an instance field. Thus its effect is an instance field write. Therefore, the effect of the `filter` method for both `Bayesian` and `Markovian` classes may be an instance field write to the same field of the same object. Thus, a static type-and-effect system may conclude that concurrently executing these two methods can lead to data races and reject this implementation as concurrency-unsafe.

However, there are many scenarios in which this application can reap concurrency benefits. For example, when no `Logger` is registered, a `Logger` is only enabled for a certain duration and then disabled, a `Logger` is registered with only one registered filter and not with all filters, etc. In other words, concurrency benefits may be sacrificed by a static type-and-effect system because the runtime costs to differentiate between these safe scenarios and the unsafe ones often overshadow gains due to concurrency.

1.3 Contribution

The main novelty of this work stems from our insight that, even though in general detecting whether a concurrency-unsafe control flow is about to run may have prohibitive costs, for implicit invocation mechanisms it can be done at an acceptable cost. This is because (a) handler registrations are infrequent compared to event announcements, and (b) the exact set of concurrent tasks (e.g. in the method `notify`) that will be run when an event is signaled and their conflicts can be computed during handler registration.

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

For example, on line 52 in Figure 1, the type-and-effect system denotes the effect of the method `filter` of class `Bayesian` to be {**`ann`** }. It assumes the **`ann`** effect does not conflict with another **`ann`** effect, thus it is safe to parallelize the `filter` methods. If however, a `logger` registers on line 53 in Figure 1, our hybrid system will dynamically enlarge the effect of the `filter` method. The effect set becomes {**`ann`** ..., **`write`** ...} and it is no longer safe to execute the `filter` methods concurrently. This is desirable, since a type system must first ensure the correctness of the program and then may endeavor to run it fast. Therefore, on registration the hybrid system (re)computes a schedule that maximizes concurrency by executing non-conflicting handlers in concurrent. Since registration is infrequent compared to event announcements, and the overhead of this dynamic effect management is small, it is amortized by the introduced concurrency.

To evaluate our approach, we have implemented our type-and-effect system, applied it to II programs, proven its key properties, and evaluated its performance benefits and overheads. Our implementation showed almost linear speedup and overheads that

ranges from 1.8 - 2.4 times the cost of registering a handler in an observer idiom (a list addition). Thus, for very small cost, our type-and-effect system provides good speedup.

In summary, this paper makes the following contributions:

- a new hybrid type-and-effect system that facilitates concurrency in shared memory programs that use implicit-invocation design style;
- a precise dynamic semantics that uses the effect system to maximize concurrency;
- a soundness proof that our system guarantees no data races and no deadlock;
- a study showing the applicability of our approach; and
- a detailed analysis of our approach and closely related ideas.

2 A Core Calculus with Implicitly Concurrent Events

We present our type-and-effect system using a calculus with support for implicitly concurrent events. The technical presentation of this calculus builds on previous calculi [19, 20]. It precisely defines language features that we have previously explored in our work on the Pāñini language [10]. Pāñini is an implicitly concurrent language, it does not feature any construct for spawning threads or for mutually exclusive access to shared memory. Rather, concurrent execution is facilitated by announcing events, using the **announce** expression, which may cause handlers to run concurrently. While previous work informally defined Pāñini [10], this calculus formalizes its definition as an expression language. Here, we describe the syntax using the example from Section 1.

```
1 class Client {
2  void receive(Email e){
3   announce Available(e);
4   //further processing details elided
5  }
6 }
7 event Available{ Email e; }
8 event SpamFound{ Email e; Filter f; }
9 class Bayesian{
10  void init(){ register(this); }
11  when Available do filter;
12  void filter(Email e){
13   if(isSpam(e))
14    announce SpamFound(e, this);
15  }
16  boolean isSpam(Email e){
17 // Use Bayesian method to detect spam
18  }
19 }
```

```
20 class Logger {
21  List l = new ArrayList();
22  void init(){ register(this); }
23  when SpamFound do act;
24  void act(Email e, Filter f){
25   l.add(e.toString()+f.toString());
26  }
27 }
28 class Markovian{
29  void init(){ register(this); }
30  when Available do filter;
31  void filter(Email e){
32   if(isSpam(e)) announce SpamFound(e, this);
33  }
34  boolean isSpam(Email e){
35 // Use Markovian method to detect spam
36  }
37 }
```

Fig. 3. An example Pāñini program that implements spam filter.

The program in Figure 3 is similar to the OO version in Figure 1, except that the code for implementing the observer pattern is replaced with Pāñini's constructs for declaring and announcing events. For example, the event type `Available`, on line 7, is used to decouple the e-mail `Client` from the concrete filters. Instead of registering with a certain client, the concrete filters register with an event. For example, on line 10, a `Bayesian` instance could dynamically register with the event `Available`. The code

for traversing the list of handlers in `Client` is replaced by an **`announce`** expression, on line 3 that notifies registered handlers. Pāñini's syntax is shown in Figure 4.

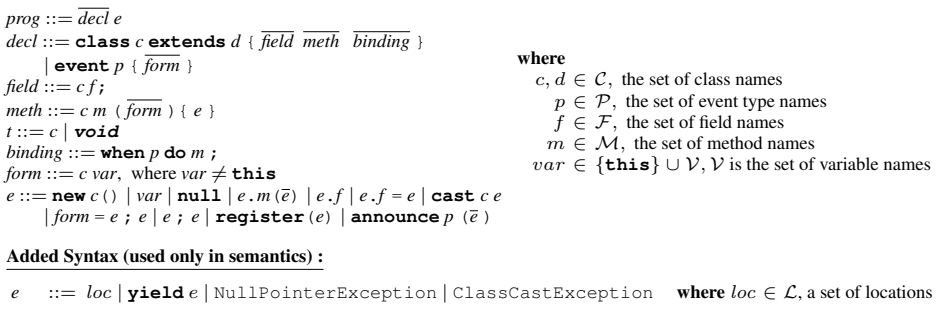

Fig. 4. Pāñini's abstract syntax, based on [20].

Top-level Declarations. Pāñini features two new declarations: event type (**`event`**) and binding declaration. An event has a name (p) and context variable declarations ($\overline{form}$). The over-bar denotes a finite ordered sequence and is used throughout this paper ($\overline{a}$ equals $a_1 \ldots a_n$). For example, in Figure 3 on line 7, an event of type `Available` is defined. It has one context variable `e` of type `Email`, which denotes the email received. These context variables are bound to actual values and made available to handlers when an event is fired. A binding declaration consists of two parts: an event type name and a method name. For example, on line 11, the class `Bayesian` declares a binding such that the `filter` method is invoked whenever an event of type `Available` is announced. This method may run concurrently with other handler methods.

Expressions. In Pāñini, handlers could register with events dynamically, for example, lines 10, 22, 29. The syntax includes standard OO expressions for object allocation, variable binding and reference, **`null`** reference, method invocation, field access and update, type casting and sequence.

Concurrency in Pāñini. The **`announce`** expression in Pāñini is the source of concurrency. The expression **`announce`** `p ( e )` announces an event of type `p`, which may run any handlers that are applicable to p *concurrently*. In Figure 3 the body of the `receive` method contains an **`announce`** expression on line 3. When the method signals this event, Pāñini looks for any applicable handlers. Here, the handlers `Bayesian` and `Markovian` are registered with the event `Available`. They may execute concurrently, depending on whether they interfere with each other. Our hybrid type-and-effect system ensures that no conflicting handlers execute concurrently (The details are in Section 3 and Section 4). After all the handlers are finished, the evaluation of the announce expression then continues on line 4. The announce expression, when signaled, binds values to the event type's context variables. For example, when announcing event `Available` on line 3, parameter `e` is bound to the context variable `e` on line 7. This binding makes the received email available to handlers in the context variable `e`.

Intermediate Expressions. Four expressions are added for the semantics, as shown in the bottom of Figure 4. The *loc* expression represents store locations. Following Abadi and Plotkin [21], we use the **`yield`** expression to model concurrency. The **`yield`** expression allows other tasks to run. Two exceptional final states, `NullPointerException` and `ClassCastException`, are reached when trying to access a field or a method from a **`null`** receiver or when an object is not a subtype of the casting type.

3 Pāñini's Type and Static Effect Computation System

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

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

Fig. 5. Type and effect attributes.

Compared to type systems that include events [20], new to our system are effects. For example, the type attributes for expressions are represented as (t, ρ), the type of an expression (t) and its effect set (ρ). The effects are used to compute the potential conflicts between handlers. These effects include: 1) read effect: a class and a field to be read; 2) write effect: content is similar to read effect; 3) announce effect: what event a certain expression may announce and 4) register effect: produced by a **`register`** expression. For example, in Figure 3, before the program runs, the effect of the method `filter` in the class `Bayesian` is {**`ann`** `SpamFound`} and the effect of the method `act` in the class `Logger` is {**`write`** `ArrayList data`} (for simplification, we assume that the method `add` in class `ArrayList` only changes the field `data`). We have intentionally avoided tracking object instances to simplify this discussion, whose focus is on event registration and announcement, however, such extension is feasible.

The interference between the effects is shown in Figure 6. Read effects do not interfere with each other. Write effects conflict with either another read or write effect accessing the same field of the same class. The announce effect is used later in the semantics. It serves as a place holder and does not conflict with other announce effects.

Announce effects will interfere with register effects, because the order of these two operations affects the set of handlers run during announcement (e.g. even if an event is fired, a handler will not get executed if it has not registered). Register effects interfere with read/write effects as well. After a handler registers with a certain event, the effect of some other handlers could be enlarged as well. Thus it could introduce a cascade of changes. Our hybrid system simply makes register effects conflict with any other effect.

Effects	`read`	`write`	`ann`	`reg`
`read`	×	√	×	√
`write`	√	√	×	√
`ann`	×	×	×	√
`reg`	√	√	√	√

Fig. 6. Effect interference. √: conflicts, ×: no conflicts

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

The type checking rules are shown in Figures 7 and 11. The notation $\nu' <: \nu$ means ν' is a subtype of ν. It is the reflexive-transitive closure of the declared subclass relationships. We state the type checking rules using a fixed class table (list of declarations CT) as in Clifton's work [19]. The class table can be thought of as an implicit inherited attribute used by the rules and auxiliary functions. We require that top-level names in the program are distinct and that the inheritance relation on classes is acyclic. The typing rules for expressions use a simple type environment, Π, which is a finite partial mapping from locations loc or variable names var to a type and an effect set.

3.1 Top-level Declarations

The rules for top-level declarations are fairly standard. The (T-PROGRAM) rule says that the entire program type checks if all the declarations type check and the expression e has any type t and any effect ρ. The (T-EVENT) rule says that an event declaration type checks if the types of all the context variables are declared properly. The (T-CLASS) rule says that a class declaration type checks if all the following constraints are satisfied. First, all the newly declared fields are not fields of its super class (this is checked by the omitted auxiliary function *validF*). Next, its super class d is defined in the Class Table. Finally, all the declared methods and bindings type check.

The (T-METHOD) rule says that a method declaration type checks if all the following constraints are satisfied: the return type is a class type (by the auxiliary function

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

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

$$\frac{CT(p) = \texttt{event}\ p\ \{t_1\ var_1, \ldots, t_n\ var_n\} \quad (c_1, t, m\ (\overline{t\ var})\ \{e\}, \rho) = \mathit{findMeth}(c, m) \quad \pi = \{var_1 : t_1, \ldots, var_n : t_n\} \quad (\forall\ (t_i\ var_i) \in \overline{t\ var} \;::\; \pi(var_i) <: t_i)}{\vdash \texttt{when}\ p\ \texttt{do}\ m \;:\; \text{OK in } c} \;\text{(T-BINDING)}$$

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

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

Fig. 7. Type-and-effect rules for declarations [19, 20].

isClass(c), shown in Figure 8, which searches CT to check whether the class c was declared. This auxiliary method is used throughout this paper); if all the parameters have their corresponding declared types, the body of the method has type u and effect ρ (Method effects ρ are stored in CT and can be retrieved by the *findMeth* function.); u is a subtype of the return type t. This rule uses an auxiliary function *override*, defined in Figure 9. It requires that the method has either a fresh name or the same type as the overridden superclass method [19]. This definition precludes overloading. In addition to standard conditions, this function enforces that the effect of an overriding method is the subset of the effect of overridden method[1].

$$\mathit{isClass}(t) = (\texttt{class}\ t \ldots) \in CT$$

Fig. 8. Auxiliary functions used in type rules, based on [20].

$$\frac{CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ldots\ meth_1 \ldots meth_p\} \quad \nexists i \in \{1..p\} \cdot meth_i = t\ m(t_1\ var_1, \ldots, t_n\ var_n)\{e\} \quad \mathit{override}(m, d, (t_1 \times \ldots \times t_n \to t, \rho))}{\mathit{override}(m, c, (t_1 \times \ldots \times t_n \to t, \rho))}$$

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

Fig. 9. Auxiliary functions used in type rules, based on [19].

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

The (T-BINDING) rule says that a binding declaration type checks if the named method is properly defined; all the context variables are subtypes of their corresponding declared types in the method; the named event type is declared properly. This rule uses the auxiliary method *findMeth*, shown in Figure 10. This method looks up the method `m`, starting from the type of the expression, looking in super classes, if necessary.

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

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

Fig. 10. Auxiliary functions used in type rules, based on [19].

3.2 Expressions

The type rules for the expressions are shown in Figure 11.

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

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

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

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

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

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

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

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

(T-NULL)
$$\frac{isClass(c)}{\Pi \vdash \texttt{null} : (c, \{\})}$$

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

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

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

Fig. 11. Type and effect rules for expressions [19, 20].

The rules for object-oriented expressions are mostly standard, except for the addition of effects in type attributes. The (T-NEW) rule ensures that the class c being instantiated was declared. This expression has type c and empty effect. The (T-GET) rule says that a field access expression returns the type of the field of the class, the effects of it will be the effect of the object expression plus a read effect. The auxiliary function *typeOfF*, shown in Figure 12, returns the type of the field. The (T-SET) rule says that a field assignment expression type checks if the object expression is of a class type and the type of the assignment expression e_2 is a subtype of the type of the field of the class. The effects will be the union of the effects of its two subexpressions plus one **`write`** effect.

$$\frac{CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{t_1\ f_1 \ldots t_n\ f_n\ \overline{meth}\ \overline{binding}\} \qquad \exists i \in \{1 \ldots n\}\ ::\ t_i\ f_i \cdot f_i = f}{typeOfF(c, f) = t_i}$$

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{t_1\ f_1 \ldots t_n\ f_n\ \overline{meth}\ \overline{binding}\} \\ \nexists i \in \{1 \ldots n\}\ ::\ t_i\ f_i \cdot f_i = f \qquad typeOfF(d, f) = t \end{array}}{typeOfF(c, f) = t}$$

Fig. 12. Auxiliary functions used in type rules, based on [19].

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 effect. The (T-SEQUENCE) rule states that the sequence expression has same type as the last expression and its effects are the union of the two expressions. The (T-YIELD) rule says that a **`yield`** expression has the same type and same effect as the expression e. The (T-VAR) rule checks that the var is in the environment. The (T-DEFINE) rule for declaration expressions says 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 should type check. The (T-NULL) rule says that the null expression will type check for no condition. It has any class type and has no effect.

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

The (T-CALL) is similar to the announce expression. This rule says that for a method call expression it finds the method in the CT using the auxiliary function `findMeth` (in Figure 10) 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.

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

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

Domains. The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 13. The rules and auxiliary functions all make use of an implicit attribute CT, the program's declarations.

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

Σ	$::= \langle \psi, \mu, \gamma \rangle$	"Program Configurations"
ψ	$::= \langle e, \tau \rangle + \psi \mid \bullet$	"Task Queue"
τ	$::= \langle n, \{n_k\}_{k \in K} \rangle$ **where** $n, n_k \in \mathcal{N}$ *and* K *is finite*	"Task Dependencies"
μ	$::= \{loc \mapsto o_k\}_{k \in K}$, **where** K *is finite*,	"Stores"
v	$::=$ **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	$::= loc + \gamma \mid \bullet$	"Handlers List"

Evaluation contexts:

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

Fig. 13. Domains, and evaluation contexts used in the semantics, based on [20].

A configuration consists of a task queue ψ, a global store μ, and a global handlers list γ. The store μ is a mapping from locations (loc) to objects (o). The handler 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 dependencies τ. This expression e serves as the remaining evaluation to be done for the task.

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

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

Evaluation Contexts. We present the semantics as a set of evaluation contexts $\mathbb{E}$ (Figure 13) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context [22]. This avoids the need for writing out standard recursive rules and clearly presents the order of evaluation. The language uses a call-by-value evaluation strategy. The initial configuration of a program with a main expression e is $\langle \langle e, \langle 0, \emptyset \rangle \rangle, \bullet, \bullet \rangle$.

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

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

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

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

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

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

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

Fig. 14. Semantics of object-oriented expressions in Pāñini, based in part on [19, 20]

Semantics for Object-oriented Expressions. The rules for OO expressions are given in Figure 14. These are mostly standard and adopted from Ptolemy's semantics [20].

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 no specific reasons for inserting the intermediate expression **`yield`** into these three expressions and not into others. We simply illustrate task interleavings by way of these three expressions and readers are encouraged to consider task interleavings in other expressions.

The (NEW) rule creates a new object and initializes its fields to null. It then creates a record with a mapping from a reference to this newly created object. The *fieldsOf* function, in Figure 15, returns a map from all the fields defined in the class and its supertypes to the types of those fields.

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

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

Fig. 15. Auxiliary functions used in semantics, based on [19].

The (CALL) rule acquires the method signature using the auxiliary function *findMeth* (defined in Figure 10). It uses dynamic dispatch, which starts from the dynamic class (c) of the record, and may look up the super class of the object if needed. The method body is to be evaluated with the arguments substituted by the actual values as well as

the `this` variable by loc. The entire substituted method body is then put inside a yield expression to model concurrency, which will be discussed later.

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

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

Semantics for Yielding Control. In Pāñini's semantics, like Abadi and Plotkin [21], the running task may implicitly relinquish control to other tasks. The rules for yielding control are given in Figure 16.

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

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

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

Fig. 16. Semantics of yielding control in Pāñini

The (YIELD) rule puts the current task configuration to the end of the task-queue and starts evaluating the next active task configuration from this queue. Finding an active task is done by the auxiliary function $active$ (shown in Figure 17). It returns the top most task configuration in the queue that could be run. A task configuration is ready to run if all the tasks in its dependent set are done (evaluated to a single value v).

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

Fig. 17. Auxiliary functions for returning a nonblocked configuration.

The (YIELD-DONE) rule is applied when there is no other task configuration in the queue. It continues to evaluate the current configuration. The (YIELD-END) rule says

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

Semantics for Event registration. We now describe the semantics for subscribing to an event (Figure 18).

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

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

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

Fig. 18. Semantics of registration and announcement

The (MULTI-REGISTER) rule is applied when the handler has already registered before ($loc \in \gamma$) and thus the configuration does not change. Pāñini does not allow multiple registrations for the same object for simplicity. The (REGISTER) rule finds out that this handler is not in the handlers list γ, so this handler is put at the front of the queue.

Semantics for announcing an event. The semantics for signaling events is shown in Figure 18. The (ANNOUNCE) rule takes the relevant event declaration from CT (the program's list of declarations) and creates a list of actual parameters (ν). This list of actual parameters (ν) is used by the auxiliary function $spawn$ shown in Figure 19 (with other helper functions in Figure 20). The (ANNOUNCE) rule resorts to the auxiliary function $spawn$ for two tasks: 1) finding the handlers registered for the corresponding event; 2) for organizing them to guarantee safety and maximize concurrency. It can then safely put the handler configurations (returned from the auxiliary function $spawn$) into the queue. The auxiliary function $concat$ is used in several other auxiliary functions. It combines the contents in the two lists, which are the inputs to this function. The first task is done by the function $spawn$. It searches the program's global list of handlers (γ) for applicable handlers, using auxiliary functions $hfind$, $hmatch$, and $match$ [20].

The second task is done by the functions $buildconfs$ (Figure 20) and $buildconf$. They create task configurations for handlers. $buildconf$ binds the context variables (of the event type) with the values (ν), computes a unique id for each handler task, and configures the dependent set of this handler. These task configurations are used to run the handler bodies and are appended to the end of the queue ψ. The auxiliary function max (shown in Figure 21) is used for giving the newly-born task a fresh ID.

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

$$spawn(p, \psi, \gamma, \nu, \mu) = buildconfs(H, \psi, \nu, \bullet, \gamma, \mu) \qquad \textbf{where } H = hfind(\gamma, p, \mu)$$

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

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

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

Fig. 19. Functions for creating task configurations.

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

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

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

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

Fig. 20. Functions for building handler configurations.

registered for events that this current task could signal, and unions their effect sets with the effect set of this task t (e.g. if t may announce an event p, then the effect of all the handlers registered for event p will be used). Pāñini does this to get more accurate information about the potential effect sets of a task to reduce false conflicts. The function *diff* (shown in Figure 21) is used to actually compare the effects to check whether they conflict with each other. The table in Figure 6 is used to compare effects.

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

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

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

$$diff(\bullet, \rho) = true$$
$$diff(\epsilon + \rho', \rho) = diff(\rho', \rho) \quad \text{where } true = differ(\epsilon, \rho)$$
$$diff(\epsilon + \rho', \rho) = false \quad \text{where } false = differ(\epsilon, \rho)$$

$$differ(\epsilon, \bullet) = true$$
$$differ(\epsilon, \epsilon' + \rho) = differ(\epsilon, \rho) \quad \text{where } \epsilon \text{ and } \epsilon' \text{ have no conflict}$$
$$differ(\epsilon, \epsilon' + \rho) = false \quad \text{where } \epsilon \text{ and } \epsilon' \text{ have conflicts}$$

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

Fig. 21. Miscellaneous helper functions.

5 Evaluation of Pāñini's Type-and-Effect System

In this section, we evaluate formal properties of our type-and-effect system as well as present an evaluation of its design and performance benefits. All performance-related experiments were run on a system with a total of 24 cores (two 12-core AMD Opteron 6168 chips 1.9GHz) running Fedora GNU/Linux. For each of the experiments, an average of the results over ten runs was taken.

5.1 Properties of Pāñini's Type-and-Effect System

Deadlock Freedom. The first property of Pāñini's type-and-effect system is that it has no deadlocks. Below we state and provide a proof sketch of this property.

Definition 1. *[Blocked Configurations.] A task configuration $\langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle$ may block if any one of its predecessors* [2] *is still in execution.*

Theorem 1. [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 dependencies, 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

[2] 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, 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 described in Section 4).

conflict with each other, then the registration order is used to create a non-blocking ordering for the handlers (discussed in Section 4). Finally, leaf nodes, which have no child dependencies, can always either be run concurrently or in an order determined by during registration. Thus, in the lowest level of the tree (leaves), there is at least one task (the handler in this level that registered earlier than any of its siblings) that does not block. Therefore, a well typed Pāñini program does not deadlock.∎

Data Races Freedom. Another property of the hybrid system is that it is free of data races. The statement and proof of this property builds on Welc's work [23].

Definition 2. *[Schedule.] A schedule (χ) is a sequence of read, write, announce and register operations performed during the evaluation of a program. More precisely, $\chi ::= \overline{\eta}$, where $\eta ::= (\texttt{rd}, n, loc, f) \mid (\texttt{wt}, n, loc, f) \mid (\texttt{an}, n, p) \mid (\texttt{rg}, n)$.*

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

$$\frac{\langle\langle \mathbb{E}[\texttt{announce}\ p\ (v_1, \ldots], \langle t, \ldots\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \Sigma \quad \chi' = \chi + (\texttt{an}, t, p)}{\chi \hookrightarrow \chi'} \qquad \frac{\langle\langle \mathbb{E}[\texttt{register}(loc)], \langle t, \ldots\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \Sigma \quad \chi' = \chi + (\texttt{rg}, t)}{\chi \hookrightarrow \chi'}$$

Definition 3. *[Schedule Safety.] A schedule χ is safe if and only if: 1) an access to a field of an object $o.f$ performed by a predecessor should not witness a write to $o.f$ by its successor[3]$(ssafe)$; (2) a write to $o.f$ by a predecessor should be visible to the first access to that field by its successor $(psafe)$; 3) an event announcement by a predecessor should not notify handlers registered by its successor $(rsafe)$ and 4) an event announcement by a successor should notify handlers registered by its predecessor $(asafe)$.*

$$\frac{(\texttt{wt}, t', loc, f), (\texttt{rd}, t', loc, f) \notin \chi' \qquad t' \leq t}{ssafe(\chi + (\texttt{wt}, t, loc, f) + \chi')} \qquad \frac{(\texttt{wt}, t', loc, f), (\texttt{rd}, t', loc, f) \notin \chi \qquad t \leq t'}{psafe(\chi + (\texttt{wt}, t, loc, f) + \chi')}$$

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

The first two conditions are roughly the same as in Welc's work [23], while the last two are necessary to ensure that handlers handle and only handle appropriate events.

Theorem 2. [Race Freedom.] *Any schedule χ produced by a Pāñini program is safe.*

Proof Sketch: (1) If tasks t and t' do not access the same field, the first two conditions in Definition 3 hold. That is because, in both $(ssafe)$ and $(psafe)$, $\nexists loc, f$, such that the conditions are violated;

(2) If they (t' and t) access the same field and at least one of them is a write, Pāñini will never schedule them to run concurrently. The (ANNOUNCE) rule and the function $buildconf$ in Section 4 ensure that no conflicting tasks should be schedule to run concurrently. Assume that these two conflicting effects are ϵ and ϵ', $differ(\epsilon, \epsilon') = false$ as is shown in Figure 21. After that $buildconf$ makes one of these tasks depends on the other (Assume t' depends on t). Finally, by the *getActive*, which is used the (*Yield*) and (*Task-End*) rules, t' can not run, until t is done.

[3] A task t is t''s successor if 1) t' is t's predecessor; or 2) handlers h' and h, corresponding to t' and t, registered to the same event and h' registered earlier than h. This is denoted as $t' \leq t$.

(3) The hybrid system makes register effects conflict with all other effects. That is, if $\epsilon' = \texttt{reg}$ or $\epsilon = \texttt{reg}$, then $\mathit{differ}(\epsilon', \epsilon) = \mathit{false}$. Therefore, the last two conditions hold. ∎

Type Soundness. The proof of soundness of Pāñini's type-and-effect system uses a standard preservation and progress argument [22]. The details are adapted from previous work [19,24]. Throughout this section we assume a fixed, well-typed program with a fixed class table, CT. A type environment $\Pi ::= \{I : \{t, \rho\}\}$ maps variables and store locations to types and effect sets. The effect set was used in the semantics to compute the dependency between handlers and will not be used in the following section. For simplicity, we omit the effect sets ρ in subsequent discussion. The key definition of consistency is as follows (the auxiliary function if defined in Figure 15).

Definition 4. *[Environment-Store Consistency.] Suppose we have a type environment Π and μ a store. Then μ* is consistent with *Π, written $\mu \approx \Pi$, if and only if all the followings hold:*

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

We now state the standard lemmas for substitution, extension, environment contraction, replacement and replacement with subtyping. These lemmas can be proved by adaptations of Clifton's proofs for MiniMAO$_0$ [19].

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

Proof Sketch: The proof proceeds by structural induction on the derivation of $\Pi \vdash e : t$ and by cases based on the last step in that derivation. The base cases are (T-NEW), (T-NULL) and (T-VAR), which have no variables and $s = t$. Other cases can be proved by adaptations of MiniMAO$_0$ [19]. The induction hypothesis (IH) is that the lemma holds for all sub-derivations of the derivation. The cases for (T-CAST), (T-SEQUENCE), (T-SET), (T-CALL) and (T-GET) are similar to Clifton's proofs. We now consider the case for (T-DEFINE), (T-REGISTER), (T-ANNOUNCE) and (T-YIELD).

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

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

The cases for $\texttt{yield}\ e$ and $\texttt{register}(e)$ are straightforward, because the type of $\texttt{yield}\ e$ and $\texttt{register}(e)$ is the same as e. ∎

Lemma 2. [Environment Extension.] *If* $\Pi \vdash e : t$ *and* $a \notin dom(\Pi)$*, then* $\Pi, a : t' \vdash e : t$*.*

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

Lemma 3. [Environment Contraction.] *If* $\Pi, a : t' \vdash e : t$ *and* a *is not free in* e*, then* $\Pi \vdash e : t$*.*

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

Lemma 4. [Replacement.] *If* $\Pi \vdash \mathbb{E}[e] : t, \Pi \vdash e : t'$*, and* $\Pi \vdash e' : t'$*, then* $\Pi \vdash \mathbb{E}[e'] : t$*.*

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

Lemma 5. [Replacement with Subtyping.] *If* $\Pi \vdash \mathbb{E}[e] : t, \Pi \vdash e : u$*, and* $\Pi \vdash e' : u'$ *where* $u' <: u$*, then* $\Pi \vdash \mathbb{E}[e'] : t'$ *where* $t' <: t$*.*

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

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

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

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

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

$$\frac{\begin{array}{c} CT(p) = \textbf{event}\ p\ \{t_1\ var_1; \ldots t_n\ var_n;\} \qquad (\forall\, i \in \{1..(p-1)\}\ ::\ \Pi \vdash v_i : t_i' \ \wedge\ t_i' <: t_i) \\ \Pi \vdash e : u \qquad u <: t_p \qquad (\forall\, j \in \{(p+1)..n\}\ ::\ \Pi \vdash e_j : t_j' \ \wedge\ t_j' <: t_j) \end{array}}{\Pi \vdash \mathbb{E}[e] : \textbf{void}}$$

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

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

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

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

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

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

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

Theorem 3. [Progress.] *For a well-typed expression e, a task dependencies τ, a task queue ψ, a store μ, and a handler queue γ. If $\Pi \vdash e : t$ and $\mu \approx \Pi$, then either $e = loc$ or $e = \textbf{null}$ 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 $e = \textbf{null}$, it is trivial.

(b) Cases $e = \texttt{NullPointerException}$ or $e = \texttt{ClassCastException}$, which are final states of the programs, result from the semantics rules $\textbf{null}.f$, $\textbf{null}.f = v$, $\textbf{null}.m(v_1, \ldots, v_n)$, $\textbf{register}(\textbf{null})$ and $\textbf{cast}\ e$. We presented the rules in Figure 22. These values serve as the base cases.

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

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

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

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

Fig. 22. Operational semantics of expressions that produce exceptions, base on, [20].

(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 $e = \mathbb{E}[\textbf{new}\ c()]$, $e = \mathbb{E}[loc.m(v_1, \ldots, v_n)]$, $e = \mathbb{E}[loc.f]$, $e = \mathbb{E}[loc.f = v]$, $e = \mathbb{E}[\textbf{cast}\ t\ loc]$, $e = \mathbb{E}[t\ var = v; e]$ and $e = \mathbb{E}[v; e_1]$ are similar to Clifton's work [19] and are omitted.

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

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

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

Theorem 4. [Subject-reduction.] *Let e be an expression and $e \neq \textbf{yield}\ e_1$ for any e_1, τ task dependencies, ψ a task queue, μ a store, and γ a handler queue. Let Π be a type environment such that $\mu \approx \Pi$. And let t a type. If $\Pi \vdash e : t$ and $\langle\langle e, \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle e', \tau'\rangle + \psi', \mu', \gamma'\rangle$, then there is some $\mu' \approx \Pi'$ and t' such that $\Pi' \vdash e' : t'$ and $t' <: t$.*

Proof Sketch: The proof is by cases on the definition of $\hookrightarrow$ separately. The cases for object oriented parts (rules (NEW), (NULL), (CAST), (GET), (SET), (VAR), (SEQUENCE) and (CALL)) can be proved by adaptations of Clifton's proofs for MiniMAO$_0$ [19].

The rule for (SEQUENCE) is similar to Clifton's work, except that $e' = \mathbb{E}[\textbf{yield}\ e]$ instead of $e' = \mathbb{E}[e]$. Since the type of $\textbf{yield}\ e$ 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 1, $\Pi \vdash [var/v]e_1 : u''$ for some $u'' <: u' <: u$. Therefore, by lemma 5, $\Pi \vdash e' : t'$ for some $t' <: t$. For the (MULTI-REGISTER) rule, $e = \mathbb{E}[\textbf{register}(v)]$ and $e' = \mathbb{E}[v]$. Let $\tau' = \tau$, $\mu' = \mu$, $\psi' = \psi$, $\gamma' = \gamma$ and $\Pi' = \Pi$. Obviously, $t' = t$. For the (REGISTER) rule, $e = \mathbb{E}[\textbf{register}(v)]$ 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 $e = \mathbb{E}[\textbf{announce}\ \ p\ \{v_1, \ldots, v_n\}]$. Let $\mu' = \mu$, $\gamma' = \gamma$, $\Pi' = \Pi$ and $t' = t$. $\textbf{void} <: c$ for any class type c. ∎

Definition 5. *[Thread-interleaving.]*
If $\langle\langle \mathbb{E}[\textbf{yield}\ e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}_1[e_1], \tau_1\rangle + \psi_1, \mu_1, \gamma_1\rangle$
$\ldots \hookrightarrow \langle\langle \mathbb{E}_n[e_n], \tau_n\rangle + \psi_n, \mu_n, \gamma_n\rangle \hookrightarrow \langle\langle \mathbb{E}[e], \tau\rangle + \psi', \mu', \gamma'\rangle$
or $\langle\langle \mathbb{E}[\textbf{yield}\ e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[e], \tau\rangle + \psi', \mu', \gamma'\rangle$,
we denote this as $\langle\langle \mathbb{E}[\textbf{yield}\ e], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow^* \langle\langle \mathbb{E}[e], \tau\rangle + \psi', \mu', \gamma'\rangle$,
where $\forall i\{1 \leq i \leq n\}\ \mathbb{E}_i[e_i] \neq \mathbb{E}[e]$.

Theorem 5. [Subject-reduction-Thread-interleaving.] *For an expression $e = \mathbb{E}[\textbf{yield}\ e_1]$, for any e_1, τ task dependencies, and ψ a task queue, μ a store and γ a handler queue. Let Π be a type environment such that $\mu \approx \Pi$. And let t a type. If*

$\Pi \vdash \mathbb{E}[\textbf{yield}\, e_1] : t$ and $\langle\langle \mathbb{E}[\textbf{yield}\, e_1], \tau\rangle + \psi, \mu, \gamma\rangle \hookrightarrow^* \langle\langle \mathbb{E}[e_1], \tau'\rangle + \psi', \mu', \gamma'\rangle$, then there is some $\mu' \approx \Pi'$ and t' such that $\Pi' \vdash \mathbb{E}[e_1] : t'$ and $t' <: t$.

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

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

If $n = 1$, $\langle\langle \mathbb{E}[\textbf{yield}\, e], \langle t, I\rangle\rangle + \psi, \mu, \gamma\rangle \hookrightarrow \langle\langle \mathbb{E}[e_1], \tau_1\rangle + \psi_1, \mu, \gamma\rangle$
$\hookrightarrow^* \langle\langle \mathbb{E}[\textbf{yield}\, e'_1], \tau_1\rangle + \psi'_1, \mu'_1, \gamma'_1\rangle \hookrightarrow \langle\langle \mathbb{E}[e], \tau\rangle + \psi'_1, \mu'_1, \gamma'_1\rangle$. And $\Pi \vdash t$. Since $\mu \approx \Pi$, $\exists t_1 \;::\; \Pi \vdash \mathbb{E}[e_1] : t_1$. By Theorem 4, $\exists t'_1, \Pi_1 \;::\; \Pi_1 \vdash \mathbb{E}[\textbf{yield}\, e'_1] : t'_1 \wedge \mu'_1 \approx \Pi_1$. Therefore, $\Pi_1 \vdash \mathbb{E}[e] : t' \wedge t' <: t$.

The (IH) is that there is some $\mu' \approx \Pi'$ and t' such that $\Pi' \vdash \mathbb{E}[e_1] : t'$ and $t' <: t$ for $\forall i \;::\; 1 \leq i \leq n$, the number of transitions. By Theorem 4 and it also true for the last transition, the claim is also true, by adding one more transition. ∎

Theorem 6. [Soundness.] *Given a program $P = \textbf{decl}_1 \ldots \textbf{decl}_n\, e$, if $\vdash P : (t, \rho)$ for some t and ρ, then either the evaluation of e diverges or else $\langle\langle e, \langle 0, \emptyset\rangle\rangle, \bullet, \bullet\rangle \hookrightarrow^* \langle\langle v, \tau'\rangle, \mu', \gamma'\rangle$ where one of the following holds for v: $v = loc$ or $v = \textbf{null}$ or $v =$ `NullPointerException` or $v =$ `ClassCastException`.*

Proof Sketch: If e diverges, then this case is trivial. Otherwise if e converges, then because the empty environment is consistent with the empty store. This case is proved by Theorem 3, Theorem 4 and Theorem 5. ∎

5.2 Real World Applications of Pāñini's Type-and-Effect System

We have enhanced the compiler for Pāñini to incorporate our type-and-effect system [10]. We now describe our experiences using this compiler on some real world programs. We have applied it to expose concurrency in two applications: a genetic algorithm library and a web crawler.

Concurrency in Genetic Algorithms. A genetic algorithm (GA) mimics the process of natural selection. These algorithms are computationally intensive and are useful for solving optimization problems [25]. The main idea is that searching for a desirable state is done by combining two *parent* states, instead of modifying a single state. An initial *generation* with n members is given to the algorithm. A *cross over* function is used to combine different members of the generation to develop the next generation. Optionally, members of the offspring may be randomly *mutated* slightly. Finally, members of the generations (or an entire generation) are ranked using a *fitness function*.

Figure 23 shows two main sub-algorithms of the GA that change a generation to produce a new generation: `CrossOver` and `Mutation`. To allow adding and removing other components in the flow of generations, this algorithm is implemented using the observer design pattern. These sub-algorithms serve as handlers. Once a new generation is produced, these handlers will be notified. In Figure 24, some other handlers are presented, as in JGAP [26]. A monitor will terminate the entire computation once certain criteria are met. A logger could log all the generations produced. Different fitness functions could be used for different purposes. A filter may be used to trim a certain

```
1 event GenReady {
2  //Reflective info. available at events
3  GenCont gct;
4 }
5 class CrossOver {
6  int prob; int max;
7  void init(){
8   register(this);
9  }
10  when GenReady do cross;
11  void cross(GenCont gct){
12   Generation g = gct.gen();
13   int gSize = g.size();
14   Generation g1 = new Generation(g);
15   // apply crossover funtion on g1;
16   if(g1.getDepth()<max && gct.done)
17    announce GenReady(new GenCont(g1,false));
18   }
19  }
20 }
```

```
21 class GenCont{
22  Generation g; boolean done;
23  GenCont(Generation g, boolean done){
24   this.g=g; this.done=done
25  }
26 }
27 class Mutation {
28  int prob; int max;
29  void init(){
30   register(this);
31  }
32  when GenReady do mutate;
33  void mutate(GenCont gct){
34   Generation g = gct.gen();
35   int gSize = g.size();
36   Generation g1 = new Generation(g);
37   // apply Mutation funtion on g1;
38   if(g1.getDepth()<max && gct.done)
39    announce GenReady(new GenCont(g1,false));
40   }
41  }
42 }
```

Fig. 23. Genetic algorithm: cross-over and mutation handlers

branch that is not likely to produce an optimum. All these other handlers are not needed all the time and may be registered in any combination to handle intermediate outputs. Therefore, it makes sense to decouple them from the main computations [1, pp. 293].

```
43 class ImprovementMonitor {
44  Generation lastG;
45  Float impValue;
46  void init(){ register(this); }
47  when GenReady do improve;
48  void improve(GenCont gct){
49   Generation g = gct.gen();
50   if(gain(lastG, g)){
51    gct.done = true;
52   }
53   lastG = g;
54  }
55  boolean gain(Generation g,
56              Generation g1) {
57  //A pure function that computes
58  //the value gained. Details elided
59  }
60 }

62 class Logger {
63  when GenReady do log;
64  void init(){ register(this); }
65  void log(GenCont gct){
66   logGen(gct.g);
67  }
68 }
```

```
69 class OffsetRemoverFitness {
70  Int preOffset;
71  void init(){ register(this); }
72  when GenReady do improve;
73  void evaluate(GenCont gct){
74   Int cur;
75   /*Computes the fitness value
76     and cur, detail omitted*/
77   preOffset = cur;
78  }
79 }
80 class Filter {
81  void init(){ register(this); }
82  when GenReady do filter;
83  void improve(GenCont gct){
84   Generation g = gct.gen();
85   Parents p=g.getParent();
86   if(progress(p, g)){
87    gct.done = true;
88   }
89  }
90  boolean progress(Parents p,
91     Generation g1){
92   /* A pure function that computes the value
93      gained. Details elided. */
94  }
95 }
```

Fig. 24. Genetic algorithm: other handlers

Generally, both the mutation and the crossover functions are computationally intensive and have no dependency on each other. Thus, executing these two functions in parallel is beneficial. The effects of both these handlers are $\{$**read** $GenCont\ done$, **ann** $GenReady\}$. (For `CrossOver`, the read effect comes from the field read on line 12 and the announce effect comes from the **announce** expression on line 17. On line 15, applying the crossover function on a new generation has no effect. That is because g1 was created on line 13 and all the changes to this local copy may not be visible to

other methods until it escapes [27] out of the method on line 13.) The static analysis approaches to conflict detection suffer from being too conservative and assume that an announce effect would be the union of the effects of all the handlers. So the effect would at least include the effects of the handlers shown in Figure 24 and these effects conflict with each other: these handlers methods have an instance field write effect and any one of them does not commute with itself [28]. Therefore, it is unsafe to concurrently execute `crossover` and `mutation`. However, not all applications will register all of these handlers and most of the applications do not. Also, almost all the fitness functions in JGAP [26], except some special cases like the one in Figure 24, are pure functions. Thus, our hybrid type-and-effect system could be very useful towards exposing concurrency in this genetic algorithm implementation.

We implemented this GA and ran two versions of it, one that registers a conflicting logger and another that does not, to observe the speedup from concurrently executing the handlers `Mutation` and `CrossOver`. In this experiment, we set the generation (or population) size to be 3000 and the depth (number of generations) to be 10. The results are shown in Figure 25. As expected, the one with no conflict shows good speedup (considering the concurrency available), while the one with conflicts was serialized. On the other hand, with a static type-and-effect system both of these scenarios would have been serialized, because the schedule produced at compile time must be serial if the handlers conflicts for any input.

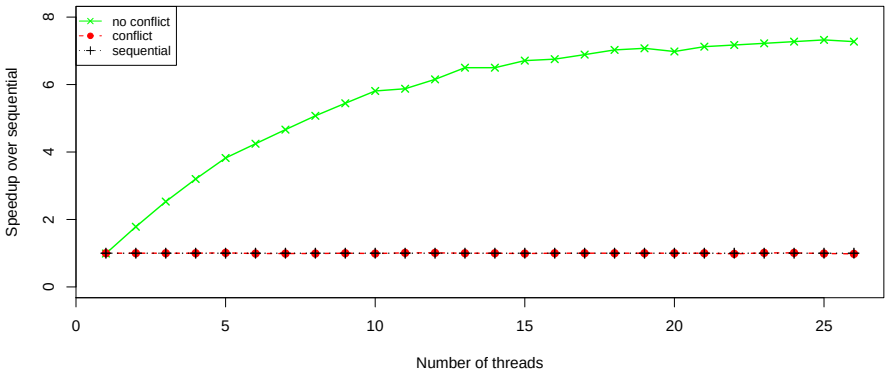

Fig. 25. Genetic Algorithm: speedup over sequential.

We also measured the overhead of registration for this application. In the experiment, the same values are used (generation size is 3000 and the depth is 10). These results are shown in Figure 26. We compared the runtime for the registrations against the runtime for the entire program. As expected, the registration takes only a very small portion of the execution time, less than 0.2%. This portion increases with more threads, because the execution time of the entire program decreases.

Applicability Beyond Implicit Concurrency. The usefulness of Pāñini's type-and-effect system is not limited to implicit concurrency. With very minor modifications, it

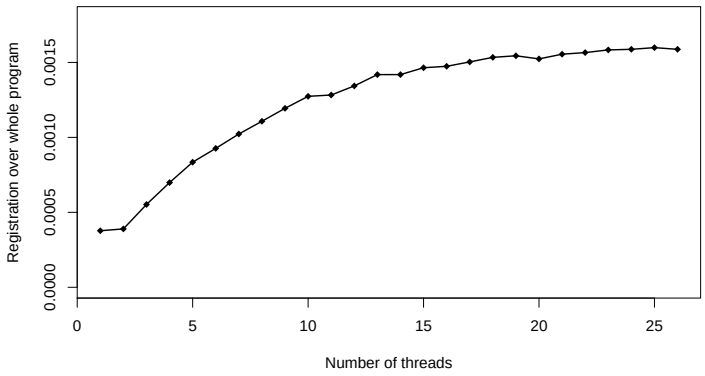

Fig. 26. Overhead of Effect System: handlers have no conflict with each other.

can be adapted to work with explicit concurrency constructs, such as ParallelFor [29] and DoAll [17]. Figure 27 and Figure 29 illustrate adaptation of the language syntax to facilitate explicit concurrency. This program implements web crawler [30], that processes web pages automatically.

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

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

Fig. 27. An example that shows the usage of the type system with explicit concurrency.

Upon receiving a link, this crawler fetches the corresponding page. It invokes classifiers to annotate pages. It expands the crawl from this page by calling the `expand` method. It will process all the links referenced to by the page until certain criteria is met, e.g. a certain depth is reached. After expanding the page (line 5), it continues processing the page (not shown). To allow future evolution of Classifiers, it is desirable that the interface `Classifier` remains independent of the `process` method.

It is beneficial to expose potential concurrency in processing links on lines 9-10. But it requires that the concurrent executions of the `process` method do not have *data races*. This could be problematic if the implementations of the interface `Classifier` write to the same memory location. It is indeed the case that in the implementation, several, but not all, concrete classes that implement this `Classifier` interface write to the same location (notice that the method does not commute with itself either [28]). However, it is beneficial to parallelize the methods that use the default classifier (not shown) which does not write to a same location, as well as other cases that use other non-conflicting classifiers. Notice that the invocation of the classifier methods is among a few operations that form the `process` method and that our hybrid system helps relieve programmers from reasoning about concurrency bugs.

We solve this problem by adding two syntactic sugars for explicitly parallel constructs `forAll` for parallel loops and `doAll` for a block of parallel statements as shown in Figure 28.

e ::= ... | `doAll` *e* `,` *e* | `forAll` `(` *var* `:` *list* `)` *e*
list ::= *e* + *list* | •

Fig. 28. Syntax for explicit concurrency that desugar to Pāñini's constructs

Here we briefly discuss the desugaring strategy for these constructs. For the parallel do of the form (`doAll` e, e'), e and e' will be evaluated in the same environment. We desugar this expression to one unique event type, two inner classes, and an announce expression. The two inner classes serve as handlers. The `doAll` expression is substituted with an announce expression (**`announce`** et), so that the two added handlers could handle this event. The parallel construct `forAll` is desugared in a similar way to a unique event type (say p'), two inner classes, and an announce expression. First inner class is a handler for the first element in the collection. The second inner class announces the same event p' with the rest of the list as the actual parameter (if the list is not empty). Thus, these two explicit concurrency constructs are syntactic sugars to the implicit concurrency constructs and no changes to the type system or the semantics rules are necessary. Snippets from the web crawler using these explicit concurrency constructs are shown in Figure 29.

```
1 event PageAvailable{ Page page; }

3 void expand (Page page) {
4   if(page.depth()>=max) return;
5   forAll(Link l: page.links()){
6     process(l); }}
```

```
7 void process (Link link) {
8   Page page = link.getPage ();
9   announce PageAvailable(page);
10  expand (page);
11 /* processing detail omitted */ }
```

Fig. 29. An example that uses explicit concurrency sugars.

This implementation is similar to the OO version, except that the for loop is replaced by `forAll` on line 5 and that traversing the list of classifiers is now delegated to the event system on line 9. We implemented this application and measured its speedup. The results are shown in Figure 30. The X-axis shows numbers of threads (amount of parallelism) and the Y-axis shows speedup over sequential version. We measured the speedup for web-crawling depths ranging from 4-8. In the experiments, the depth denotes different workloads, the deeper the depth the higher the workload. As expected, the implementation shows good speedup for reasonable workload (>1 depth).

We also measured the overhead of the type-and-effect system for this program. The results of this measurement are shown in Figure 31. The X-axis in this chart shows web-crawling depth and the Y-axis shows overhead of the effect system over the entire program's execution time. Notice that even at smaller web-crawling depths, the benefits of our type-and-effect system can be observed as the overhead is a very small portion of total execution time. For larger depths the overhead becomes negligible.

Advantages. For both genetic algorithm framework and the web-crawler application, we saw the following benefits of our type-and-effect system.

1. It is capable of detecting cases where there are no conflicts and parallelize the execution, whereas static type-and-effect systems may not.

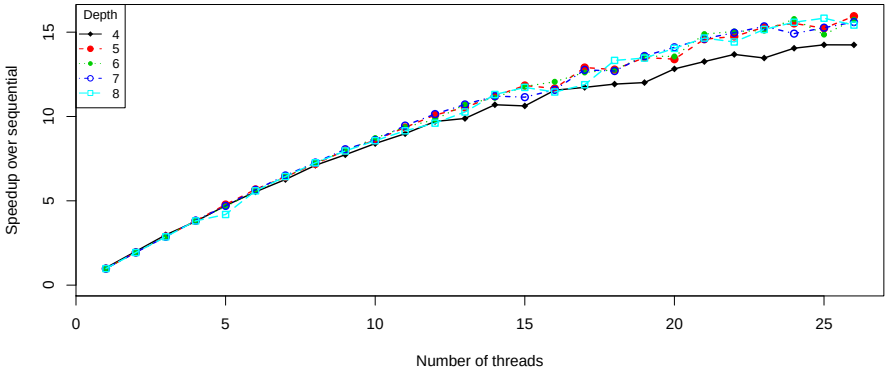

Fig. 30. Web crawler: average speedup.

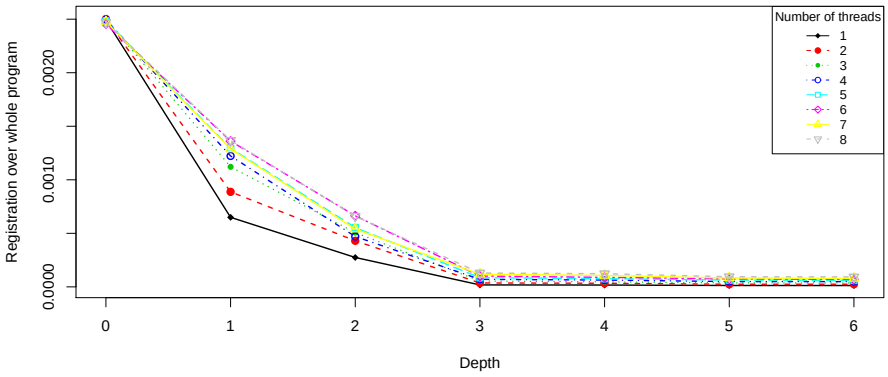

Fig. 31. Web crawler: cost of effect system.

2. It is effective at detecting conflicts between handlers and serializes these cases to prevent concurrency bugs. What is more, the overhead of doing so is small. For example, in Figure 25, the lines representing the version with conflict and the sequential OO version almost overlap with each other.
3. Programmers are relieved from reasoning about the absence of concurrency bugs.
4. To transform the OO implementation to Pāñini's version only changes necessary were to substitute the observer pattern implementation with Pāñini's constructs.

Thus, invasive modifications are avoided that are generally common in transforming a shared memory program to disjoint memory program.

5.3 Performance Evaluation of Effect System in Isolation

In this subsection, we measure the overhead introduced by our hybrid system. We compared Pāñini's implementation to an OO implementation that puts the handlers into a list. In the first experiment, we varied the number of handlers, from 1 to 45, with none of them interfering with each other. All of these handlers will be put in the first level of the hierarchy[4]. The results are shown in Figure 32.

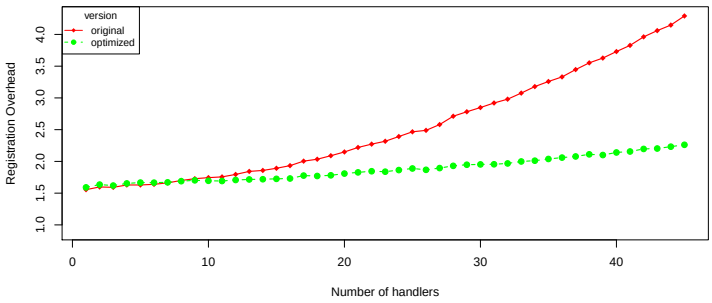

Fig. 32. Overhead of effect system: handlers have no conflict with each other. This data is normalized with respect to the overhead of the OO Observer Idiom

In this experiment, handlers registered one by one for a single event. The result shows the overhead normalized to the OO implementation of the observer design pattern, which puts handlers into a list. The line with square is the overhead of Pāñini's original implementation. We will explain the line with circles later. As expected, overhead increases with the number of handlers. A somewhat surprising observation was that with a moderate number of handlers, the overhead is acceptable (*around 1.5 to 5 times the cost of a list addition operation*).

Next, we measured the overhead when some handlers have conflicts with each other. We compared our system against the OO version, which adds handlers to a list. We fixed the number of handlers to 25. We varied the number of pairs of handlers that have conflicts, from 0 to 25. These results are shown in Figure 33.

The X axis shows the number of conflicts: 0 means no conflict; while n ($n > 0$) means that the first n handlers conflict with each other. The Y axis shows the overhead normalized to the OO version. The line with square is the overhead of our initial implementation. Somewhat surprisingly the overhead drops with more conflicts. The reason is that during registration, a handler has to be compared against the effects of the handlers in the last level (i.e. the more handlers in the last level, the more overhead it could incur). To optimize this case, we introduced a version of the system that caches the

[4] Pāñini's implementation refers to the data structure, that holds the registered handlers, as a hierarchy [10]. Handlers that do not have conflicts will be put in the same level. The first handler that has conflicts with any previous registered handler will be put in the second level etc.

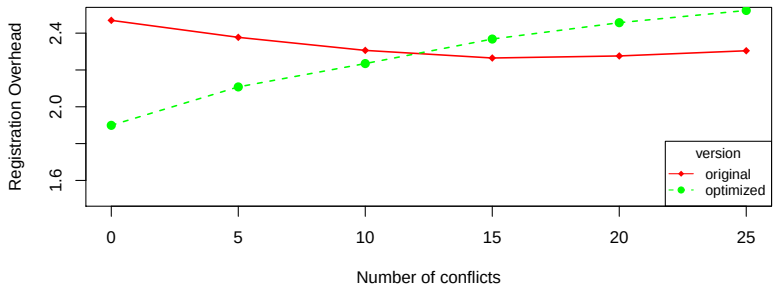

Fig. 33. Overhead of effect system: handlers may conflict with each other. This data is normalized with respect to the overhead of the OO observer idiom

effects for each level. The results are shown as the lines with dots in both figures. We observe that due to this caching mechanism, the overhead of our system drops quite a bit with no conflicts (Figure 32). In the case of conflicting handlers, the overhead increases slightly due to the overhead introduced by the caching mechanism (Figure 33).

In summary, the overhead introduced by our technique is small, and its concurrency benefits significant. It is conceivable that with some optimizations, the overhead can be reduced and speedup increased. These results thus serve to show the potential utility of our technique towards exposing implicit concurrency in shared-memory II programs.

6 Related Work

Types, Regions and Effects. Pāñini's hybrid system is not the first to use type-and-effect to enable safe concurrency. Deterministic parallel Java (DPJ/DPJizer) [17, 31] uses a region-based type-and-effect system to provide deterministic parallelism in imperative, OO programs. Ownership systems [32–34] have been used to organize objects into hierarchies for better reasoning, i.e. about the absence of aliasing. Concurrent Revisions [18] provides users with a syntax that says each thread accesses its own version of certain objects to eliminate interferences. Similar to these analyses, our hybrid system generates static invariants, e.g. effect summaries for every method. To the best of our knowledge, compared to these related ideas, Pāñini's type-and-effect system is the first system that uses the effect in a hybrid manner. The schedule produced by the purely static approaches must be valid for all inputs and thus may declare many programs concurrency-unsafe, even though only certain rare control flow paths in such programs produce concurrency-unsafe computational effects. Our hybrid system computes schedules during program execution, upon which it has more accurate information. Therefore, it could observe more safe parallelization opportunities than the purely static approaches. Also, the overhead induced by this dynamic phase is small.

Atomicity. Several systems provide syntax to declare and validate the atomicity of certain data structures and thus guarantee concurrency safety. AJ2 and Rcc/Sat [35, 36] use the type system to enforce certain locking discipline to check for data races or ensure atomic access objects. AJ [37] uses a type system to maintain data-centric synchronization and is a variant of the atomicity protection. Unlike these works, our hybrid system ensures a deterministic semantics, not just atomicity. Sometime, atomicity is not enough to guarantee a deterministic semantics. For example, although it is safe to

concurrently add elements to a list, the order of the insertions is violated and could be arbitrary. Secondly, our focus is on producing or validating a safe schedule, while they offers constructs for programmers to facilitate the reasoning on concurrency safety, statically. Therefore, these works are orthogonal to our work, thus our hybrid system may enhance the accuracy by incorporate with these static technique.

Dynamic Approaches. Dynamic approaches are also used to ensure concurrency safety. The Galois system [38, 39] aims to optimistically parallelize irregular applications. Central to this system is a worklist where pending operations live and more operations can be inserted into this list. Their underlying implementation uses speculative execution. In contrast to this, our system infers the computational effects for operations statically and then uses it at runtime to determine a safe schedule for operation execution that maximizes concurrency. So Galois system requires use of a thread speculation infrastructure at runtime, e.g. to implement a rollback mechanism, whereas our effect system requires an effect manipulation and a scheduling mechanism. Furthermore, unlike previous work on dynamic approaches [23, 40], our work does not require modifications to the underlying virtual machine.

Actor-based Languages. There is a large body of work on using the notion of actors [41] for concurrency. Agha and Hewitt's work [42] and Erlang's language design [43] model programs as a set of "isolated" active entities that communicate by passing messages. JCoBox [44] unifies the actor model with the shared memory model to enhance local and distributed concurrency. In these models, actors process local computations concurrently with other actors. The actor model is seen as naturally supporting concurrency. Also, complete isolation of actors makes it easier to reason about their states. However, in mainstream object-oriented languages such as Java, C++, etc., programmers rely on shared states to express many useful computational idioms. So although in principle it would be sensible to adopt a fully-isolated actor-based model, practice and existing investment in mainstream languages demand a solution that supports both message-passing and shared states. Also, our system guarantees a deterministic semantics, which is somewhat difficult for the actor model, due to asynchronous and non-deterministic nature of the message passing paradigm [45].

Event-based Systems. Events have a long history in both the software design [8, 9] and distributed systems communities [46]. Pāñini's notion of asynchronous, typed events builds on these notions, in particular recent work in programming languages focusing on event-driven design [4, 20, 47]. Pāñini's design is not the first to integrate event-based model with concurrency. Reactor [5] pattern integrates the demultiplexing of events and the dispatching of the corresponding event handlers to simplify event-driven applications. Li and Zdancewic [9] promote the integration of event-based model with the thread-based explicit concurrency models. TaskJava [8] provides syntax to mark `asynchronous` methods. Expressions may express their interests in a set of events and the expressions will block until one of them fires. Pāñini's design is also not the first to promote implicit concurrency, e.g. in BETA [48], objects implicitly execute in the context of a local process.

The above models, developed for event-based distributed systems, assume that components in the system do not share state and only communicate by passing primitive values, whereas Pāñini allows shared states (similar to mainstream languages like Java,

C#), which is useful for many computation patterns. Also, unlike the above works on shared memory [8, 9], Pāñini provides safety guarantee. Pāñini provides programmers with deterministic semantics via its hybrid type-and-effect system. As a result, programmers are relieved of reasoning about concurrency bugs. Such software engineering properties are becoming very important with the increasing presence of concurrent software, increasing interleaving of threads in concurrent software, and increasing number of under-prepared software developers writing code using concurrency unsafe features.

7 Conclusion and Future Work

Language features that promote concurrency for implicit invocation (II) design style in a safe manner have become important [3–5]. This idiom is widely used in mainstream shared-memory languages, e.g. via the observer design pattern [1]. Static type-and-effect systems [17, 33] are very effective at eliminating data races and deadlocks in explicitly concurrent languages, however, they are often too conservative and reject programs written in II style where concurrency could be safely exposed. The actor model [41] exposes concurrency by providing a disjoint memory model. However, due to the asynchronous nature of the message passing model, it does not provide a deterministic semantics. Also, programmers that are well-versed in mainstream OO languages, have to make great efforts to adapt to this model. The hybrid type-and-effect system proposed in this work solves these problems. We have shown several examples of using our type-and-effect system. The results gathered from running these examples have shown that the overhead of our effect system is acceptable and its performance benefits are promising. Our effect system provides race and deadlock freedom and a deterministic semantics. Last but not least, it exposes opportunities for concurrency that may be considered unsafe by completely static type-and-effect systems.

In this work, we have deliberately avoided aliasing issues by keeping the read and write effects limited. This allowed us to focus on the announcement and registration effects. In future, we would explore the integration of our type-and-effect system with an ownership type system [32–34] to further enhance its precision.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
2. Garlan, D., Notkin, D.: Formalizing design spaces: Implicit invocation mechanisms. In: VDM '91. (1991) 31–44
3. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus: An Architecture for Extensible Distributed Systems. In: SOSP. (1993) 58–68
4. Eugster, P.: Type-Based Publish/Subscribe: Concepts and Experiences. ACM Trans. Program. Lang. Syst. **29**(1) (2007) 6
5. Schmidt, D.C.: Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching. Pattern languages of program design (1995) 529–545
6. Cunningham, R., Kohler, E.: Tasks: language support for event-driven programming. In: Conference on Hot Topics in Operating Systems, Berkeley, CA, USA (2005)

7. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for c#. ACM Trans. Program. Lang. Syst. **26** (September 2004) 769–804
8. Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: PEPM. (2007) 134–143
9. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: PLDI. (2007) 189–199
10. Long, Y., Mooney, S.L., Sondag, T., Rajan, H.: Implicit invocation meets safe, implicit concurrency. In: GPCE. (2010)
11. Event-based Asynchronous Pattern. `http://msdn.microsoft.com/en-us/library/wewwczdw.aspx/`
12. Sahami, M., Dumais, S., Heckerman, D., Horvitz, E.: A bayesian approach to filtering junk e-mail (1998)
13. Chhabra, S., Yerazunis, W.S., Siefkes, C.: Spam filtering using a markov random field model with variable weighting schemas. In: ICDM. (2004) 347–350
14. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: LFP. (1986) 28–38
15. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Inf. Comput. **111** (1994) 245–296
16. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA. (2002) 211–230
17. R. Bocchino *et al.*: A type and effect system for deterministic parallel java. In: OOPSLA. (2009) 97–116
18. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA. (2010) 691–707
19. Clifton, C., Leavens, G.T.: MiniMAO$_1$: Investigating the semantics of proceed. Science of Computer Programming **63**(3) (2006) 321–374
20. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: ECOOP. (2008) 155–179
21. Abadi, M., Plotkin, G.: A model of cooperative threads. In: POPL. (2009) 29–40
22. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (Nov 1994) 38–94
23. Welc, A., Jagannathan, S., Hosking, A.: Safe Futures for Java. In: OOPSLA. (2005) 439–453
24. Flatt, M., Krishnamurthi, S., Felleisen, M.: A Programmer's Reduction Semantics for Classes and Mixins. In: Formal Syntax and Semantics of Java. (1999) 241–269
25. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edn. Prentice Hall (2003)
26. Meffert, K.: JGAP - Java Genetic Algorithms and Genetic Programming Package. `http://jgap.sf.net`
27. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA, ACM (1999) 187–206
28. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis framework for parallelizing compilers. In: PLDI. (1996) 54–67
29. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: OOPSLA. (2009) 227–242
30. Miller, R.C., Bharat, K.: Sphinx: a framework for creating personal, site-specific web crawlers. In: WWW. (1998) 119–130
31. Vakilian, M., Dig, D., Bocchino, R., Overbey, J., Adve, V., Johnson, R.: Inferring method effect summaries for nested heap regions. In: ASE. (2009) 421–432
32. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA. (2002) 292–310

33. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In: OOPSLA. (2007) 441–460
34. Cameron, N., Noble, J., Wrigstad, T.: Tribal ownership. In: OOPSLA. (2010) 618–633
35. Flanagan, C., Freund, S.N.: Type inference against races. Sci. Comput. Program. **64** (2007) 140–165
36. Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI. (2005) 47–58
37. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: A type system for data-centric synchronization. In: ECOOP. (2010) 304–328
38. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI. (2007) 211–222
39. Méndez-Lojo, M., Mathew, A., Pingali, K.: Parallel inclusion-based points-to analysis. In: OOPSLA. (2010) 428–443
40. Pratikakis, P., Spacco, J., Hicks, M.: Transparent Proxies for Java Futures. In: OOPSLA. (2004) 206–223
41. Agha, G.: Actors: a model of concurrent computation in distributed systems. Technical Report AITR-844 (1985)
42. Agha, G., Hewitt, C.: Concurrent programming using actors: Exploiting large-scale parallelism. In: Foundations of Software Technology and Theoretical Computer Science, Springer (1985) 19–41
43. Armstrong, J., Williams, R., Virding, M., Wikstroem, C.: Concurrent Programming in ERLANG. Prentice-Hal (1996)
44. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: ECOOP, Springer (June 2010) 275–299
45. Mackay, P.: Why has the actor model not succeeded?
46. Eugster, P.T., Guerraoui, R., Damm, C.H.: On Objects and Events. In: OOPSLA. (2001) 254–269
47. Eugster, P., Jayaram, K.R.: EventJava: An Extension of Java for Event Correlation. In: ECOOP. (2009) 570–584
48. Shriver, B., Wegner, P.: Research directions in object-oriented programming (1987)