Summer 7-31-2015

Design, Semantics and Implementation of the Ptolemy Programming Language: A Language with Quantified Typed Events

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Gary T. Leavens
University of Central Florida

Design, Semantics and Implementation of the Ptolemy Programming Language: A Language with Quantified Typed Events

HRIDESH RAJAN
Iowa State University
GARY T. LEAVENS
University of Central Florida

Implicit invocation (II) and aspect-oriented (AO) languages provide software designers with related but distinct mechanisms and strategies for decomposing programs into modules and composing modules into systems. II languages have explicitly announced events that run registered observer methods. AO languages have implicitly announced events that run method-like but more powerful advice. A limitation of II languages is their inability to refer to a large set of events succinctly. They also lack the expressive power of AO advice. Limitations of AO languages include potentially fragile dependence on syntactic structure that may hurt maintainability, and limits on the available set of implicit events and the reflective contextual information available. Quantified, typed events, as implemented in our language Ptolemy, solve all these problems. This paper describes Ptolemy and explores its advantages relative to both II and AO languages.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages

Additional Key Words and Phrases: aspect-oriented programming languages, quantification, obliviousness, fragile pointcuts, modular reasoning, joinpoint, context exposure, type checking.

1. INTRODUCTION

For temperance and courage are destroyed both by excess and defect, but preserved by moderation. – Aristotle, Nicomachean Ethics

The objective of both implicit invocation (II) [Luckham and Vera 1995; Notkin et al. 1993] and aspect-oriented (AO) [Kiczales et al. 1997] languages is to improve a software engineer's ability to separate conceptual *concerns*. The problem that they address is that not all concerns are amenable to modularization by a single dimension of decomposition [Tarr et al. 1999]; instead, some concerns cut across the main dimension of decomposition. The II and AO approaches aim to better encapsulate such crosscutting concerns and decouple them from other code, thereby easing software maintenance tasks.

However, both II and AO languages suffer from various limitations. The goal of this article is to explain how our language Ptolemy, which combines the best ideas of both kinds of language, can solve many of these problems.

This article is the revised and significantly extended version of the work presented at ECOOP 2008 in Paphos, Cyprus [Rajan and Leavens 2008]. This research was supported in part by NSF grants CNS-06-27354 and CNS-08-08913.
Authors' address: H. Rajan, hridesh@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010. G. Leavens, leavens@eecs.ucf.edu, School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL, 32816.

DOI: 0000001.0000001

1.1. Implicit Invocation Languages and their Limitations

The key idea in II languages is that *events* are used as a way to interface two sets of modules, so that one set can remain independent of the other. Events promote decoupling and can be seen as direct linguistic support for the Observer pattern [Gamma et al. 1995]. The mechanisms of an II language are also known as "event subscription management." [Luckham and Vera 1995]

With declared events, certain modules (subjects) dynamically and explicitly *announce* events. Another set of modules (observers) can dynamically *register* methods, called *handlers*. These handlers are invoked (implicitly) when events are announced. The subjects are thus independent of the particular observers.

Figure 1 illustrates the mechanisms of a hypothetical Java-like II language, similar to Ptolemy, for a figure editor that we will use as a running example. This code is part of a larger editor that works on drawings comprising points, lines, and other such figure elements [Gamma and Eggenschwiler 1998]. The code announces two kinds of events, named `ChangeEvent` and `MoveUpEvent` (lines 2–4). The subclass `Point` announces these events using **`announce`** expressions (lines 10, 14, and 19). When an instance of the class `Update` is properly "registered", by calling the `registerWith` method on an instance of the `Point` class, these announcements will implicitly invoke the methods of class `Update` (lines 22–41). The connection between the events and methods of class `Update` is made on lines 39–40, where it is specified that the `update` method is to be called when the `ChangeEvent` occurs and the `check` method when `MoveUpEvent` occurs. Dynamic registration (lines 25–26) allows the receiver of these method calls to be determined (and allows unregistration and multiple registration).

```
1 class FElement extends Object{
2  event ChangeEvent(FElement changedFE);
3  event MoveUpEvent(FElement targetFE,
4        Number y, Number dy);
5 }
6 class Point extends FElement{/*...*/
7  Number x; Number y;
8  FElement setX(Number x) {
9   this.x = x;
10  announce ChangeEvent(this);
11  this
12 }
13 FElement moveUp(Number dy) {
14  announce MoveUpEvent(this,this.y,dy);
15  this.y = this.y.plus(dy); this
16 }
17 FElement makeEqual(Point other) {
18  other.x = this.x; other.y = this.y;
19  announce ChangeEvent(other); other
20 }
21 }
```

```
22 class Update extends Object { /* ... */
23  FElement last;
24  Update registerWith(FElement fe) {
25   fe.register(this, FElement.ChangeEvent);
26   fe.register(this, FElement.MoveUpEvent);
27   this
28  }
29  FElement update(FElement changedFE, Number x){
30   this.last = changedFE;
31   Display.update();
32   changedFE
33  }
34  FElement check(FElement targetFE,
35     Number y, Number dy) {
36   if (dy.lt(100)) { targetFE }
37   else{throw new IllegalArgumentException()}
38  }
39  when FElement.ChangeEvent do update
40  when FElement.MoveUpEvent do check
41 }
```

Fig. 1. Drawing Editor in an II language.

The main advantage of the II programming model over the OO one is that the II programming model provides considerable automation of the Observer pattern [Luckham and Vera 1995], which is key to decoupling subject modules from observer modules. That is, modules that announce events remain independent of the modules that register methods to handle their event announcements. Compared to AO languages, as we will see, II languages also have some advantages. First, event announcement is explicit, which helps in understanding the module announcing the event, since the points where events may occur are obvious from the code. Second, event announcement is flexible; i.e., arbitrary points in the program can be exposed as events.

However, compared with AO languages, II languages also have three limitations: coupling of observers to subjects, no ability to replace event code, and lack of quantification. We describe these below.

1.1.1. Coupling of Observers to Subjects. While subjects need not know about observers in an II language, the observer modules still know about the subjects. In Figure 1, for example, the registration code on lines 25–26 and the binding code on lines 39–40 mentions the events declared in `FElement`. (Mediators, a design style for II languages, also decouple subjects and observers so that they can be changed independently [Sullivan and Notkin 1992]. However, mediator modules remain coupled to both the subject and observers.)

1.1.2. No Replacement of Event Code. The ability to replace the code for an event (what AO calls "around advice"), is not available, without unnecessarily complex code that essentially simulates around advice. Instead, to stop an action, one must have a handler throw an exception (as on line 37), which does not clearly express the idea, and does not work when the notification is done after the action has happened. Similarly, throwing an exception does not support replacing actions with different actions, such as replacing a local method call with a remote method invocation.

1.1.3. No Quantification. In II languages describing how each event is handled, which following the AO terminology we call *quantification*, can be tedious. Indeed, such code can grow in proportion to the number of objects from which implicit invocations are to be received. For example, to register an `Update` instance `u` to receive implicit invocations when events are announced by both a point `p` and a line `l`, one would write the following code: `u.registerWith(p); u.registerWith(l)`. One can see that such registration code has to find all figure element instances. In this case these problems are not too bad, since all such instances have types that are subtypes of `FElement`, where the relevant events are declared. However, if the events were announced in unrelated classes, then the registration code (lines 25–26) and the code that maps events to method calls (lines 39–40) would be longer and more tedious to write.

1.2. Aspect-Oriented Languages and their Limitations

In AO languages [Kiczales et al. 1997; Mezini and Ostermann 2003; Rajan and Sullivan 2003a] such as AspectJ [Kiczales et al. 2001] events (called "join points") are pre-defined by the language as certain kinds of standard actions (such as method calls) in a program's execution. (We emphasize AspectJ for the maturity of its design and the availability of a workable implementation.) AO events are all implicitly announced. *Pointcut descriptions (PCDs)* are used to declaratively register handlers (called "advice") with sets of events. Using PCDs to register a handler with an entire set of events, called *quantification* [Filman and Friedman 2000], is a key idea in AO languages that has no counterpart in II languages. A language's set of PCDs and events form its *event model* (in AO terms this is a "join point model").

The listings in Figure 2 shows an AspectJ-like implementation for the drawing editor discussed before. (We have adapted the syntax of AspectJ to be more like our language Ptolemy, to make comparisons easier.) In this implementation the `Point` class is free of any event-related code (as are other figure elements such as `Line`). Modularization of display update is done with an aspect. This aspect uses PCDs such as **`target`**`(fe) &&` **`call`**`(FElement+.set*(..))` to select events that change the state of figure elements. This PCD selects events that call a method matching `set*` on a subtype of `FElement` and binds the context variable `fe` (of type `FElement`) to that call's receiver.

AO languages also have several advantages. Quantification provides ease of use. For example, one can select events throughout a program (and bind them to handlers) by just writing a simple regular expression based PCD, as on lines 17–18. Moreover, by not referring to the names in the modules announcing events directly, the handler code remains, at least syntactically, independent of that code. Implicit event announcement both automates and further decouples the two sets of modules, compared with II languages. This property, sometimes called *obliviousness* [Filman and Friedman 2000], avoids the "scattering" and "tangling" [Kiczales et al. 1997] of event announcement code within the other code for the subjects, which can be seen in lines 10, 14, and 19 of Figure 1. In that figure, this explicit announcement code is mixed in with other code, resulting in tangled code that makes it harder to follow the main program flow.

```
1 class FElement extends Object{}
2 class Point implements FElement{/*..*/
3  Number x; Number y;
4  FElement setX(Number x) {
5   this.x = x; this
6  }
7  FElement moveUp(Number dy) {
8   this.y = this.y.plus(dy); this
9  }
10 FElement makeEqual(Point other) {
11  other.x = this.x;
12  other.y = this.y; other
13 }
14 }
```

```
15 aspect Update {
16  FElement around(FElement fe) :
17   call(FElement+.set*(..)) && target(fe)
18   || call(FElement+.makeEq*(..)) && args(fe){
19   FElement res = proceed(fe);
20   Display.update(); res
21  }
22 FElement around(FElement fe, Number dy):
23    target(fe)&&(call(FElement+.move*(..))
24    && args(dy){
25   if (dy.lt(100) { proceed(dy) }
26   else { fe }
27  }
28 }
```

Fig. 2. Drawing editor's AO implementation.

However, AO languages suffer from four limitations, primarily because most current event models use PCDs based on pattern matching. These languages differ by what they match. For example, AspectJ-like languages use pattern matching on names [Kiczales et al. 2001], LogicAJ and derivative languages use pattern matching on program structures [Rho et al. 2006; Eichberg et al. 2004], and, history-based pointcuts use pattern matching on program traces [Douence et al. 2004]. An example name-matching PCD is `call(FElement+.set*(..))` that describes a set of call events in which the name of the called method starts with "`set`". An example structure-matching PCD is `stmt(?if,if(?call){??someStatements}&&fooBarCalls(?call)` that describes a set of call events in which the name of the called method is "`foo`" or "`bar`" and the call occurs within an **`if`** condition [Rho et al. 2006, Fig 4.]. An example trace-matching PCD is $G(call(*Line.set(..)) \rightarrow F(call(*Point.set(..))))$ that describes every call event in which the name of the called method is "`Line.set`" and that is finally followed by another call event in which the name of the called method is "`Point.set`" [Stolz and Bodden 2005, Fig 3.].

1.3. Fragile Pointcuts

The fragility of pointcuts results from the use of pattern matching as a quantification mechanism [Stoerzer and Graf 2005; Tourwé et al. 2003]. Such PCDs are coupled to the code that implements the implicit events they describe. Thus, seemingly innocuous changes break aspects. For example, for languages that match based on names, a change such as adding new methods that match the PCD, such as `settled`, can break an aspect that is counting on methods that start with "`set`" to mean that the object is changing. As pointed out by Kellens *et al.* [Kellens et al. 2006], in languages that match based on program structures a simple change such as changing an **`if`** statement to an equivalent statement that used a conditional (`?:`) expression would break the aspect. For languages that match based on program traces a simple change such as to inline the functionality of "`Point.set`" would break the aspect that is counting on "`Line.set`" to be eventually followed by "`Point.set`" [Kellens et al. 2006]. Conversely, when adding a method such as `makeEqual` that does not conform to the naming convention, one has to change the PCD to add the new method pattern (as shown in line 18 of Figure 2). In the same vein, when adding a new call such as `foo()` within a **`while`** statement that does not conform to the existing program structure, one has to change the PCD to accomodate the new program structure. Similar arguments apply for trace-based pointcuts. Indeed, to fix such problems PCDs must often be changed (e.g., to exclude or include added methods). Such maintenance problems can be important in real examples.

Several ideas such as Aspect Aware Interfaces (AAIs) [Kiczales and Mezini 2005], Crosscut Programming Interfaces (XPIs) [Sullivan et al. 2005; W. G. Griswold et al. 2006], Model-based Pointcuts [Kellens et al. 2006], Open Modules (OM) [Aldrich 2005], etc, have recognized and proposed to address this fragile pointcut problem. Briefly, AAIs, computed using the global system configuration, allow a developer to see the events in the base code quantified by a PCD, but do not help with reducing the impact of base code changes on PCDs, which primarily causes the fragile pointcut problem. XPIs reduce the scope of fragile pointcut problem to the scope declared as part

of the interface, however, within a scope the problem remains. OMs allow a class to explicitly expose the set of events, however, for quantifying such events explicit enumeration is needed, which couples the PCD with names in the base code. Such enumerations are also potentially fragile as pointed out by Kellens *et al.* [Kellens et al. 2006]. A detailed discussion of these ideas is presented in Section 6.

1.4. Quantification Failure

The problem of quantification failure is caused by incompleteness in the language's event model. It occurs when the event model does not implicitly announce some kinds of events and hence does not provide PCDs that can select such events [Sullivan et al. 2005, pp. 170]. In AspectJ-like AO languages there is a fixed classification of potential event kinds and a corresponding fixed set of PCDs. For example, some language features, such as loops or certain expressions, are not announced as events in AspectJ and have no corresponding PCDs.[1] While there are reasons (e.g., avoiding coupling) for not making some kinds of potential events available, some practical use cases need to handle them [Harbulot and Gurd 2006; Rajan and Sullivan 2005a]. This fixed set of event kinds and PCDs contributes to quantification failure, because some events cannot be announced or used in PCDs.

There are approaches, such as LogicAJ, that provide a finer-grained event model [Rho et al. 2006]. For example, in LogicAJ one could match arbitrary program structure in the base code, which is significantly more expressive compared to matching based on names. However, as discussed above, a problem with such technique is that the PCDs becomes strongly coupled with the structure of the base code and therefore become more fragile.

An alternative approach to solving this problem is taken by the technique used in SetPoint [Altman et al. 2005]. This technique allows a programmer to select events by attaching annotations to locations of such events. This technique is not fragile in the sense that it does not depend on names, program structure, or order of events. A problem, however, is that this technique does not allow arbitrary expressions to be selected, primarily because the underlying languages do not allow annotations on arbitrary expressions.

1.5. Limited Access to Context Information

Current AO languages provide a limited interface for accessing contextual (or reflective) information about an event [Sullivan et al. 2005]. For example, in AspectJ, a handler (advice) can access only fixed kinds of contextual information from the event, such as the current receiver object (**`this`**), a call's target, its arguments, etc. Again there are good reasons for limiting this interface (e.g., avoiding coupling), but the fundamental problem is that, in current languages, this interface is fixed by the language designer and does not satisfy all usage scenarios. For example, when modularizing logging, developers need access to the context of the logging events, including local variables. However, local variables are not available in existing AO event models.

Approaches such as LogicAJ [Rho et al. 2006] allow virtually unlimited reflective access to the program context surrounding code using meta-variables, which is more expressive than AspectJ's model; e.g., a local variable can be accessed by associating it with a meta-variable. However, as we discuss in detail below, this unlimited access is achieved with ease only in cases where the events form a regular structure.

1.6. Uniform Access to Irregular Context Information

A related problem occurs when contextual information that fulfills a common need (or role) in the handlers is not available uniformly to PCDs (and handlers). For example, in Figure 2 `setX` and

[1] Some may view that as a problem of the underlying language rather than the approach to aspects: e.g., in a language where all computation takes place in methods, this, target and args are always defined. We argue that it may not be necessary to continue to support such differentiation between means of computation, instead a unified view of all such means of computation can be provided to the aspects.

`makeEqual` contribute to the event "changing a figure element," however, they are changing different figure element instances: **`this`** and `other` in the case of `setX` and `makeEqual` respectively. In this simple case, it is possible to work around this issue by writing a PCD that combines (using `||`, as in lines 17–18 of Figure 2) two separate PCDs, as shown in Figure 2. Each of these PCDs accesses the changed instance differently (one using **`target`**, the other using **`args`**). However, each such PCD depends on the particular code features that it needs to access the required information.

This problem is present in even significantly more expressive approaches based on pattern matching such as LogicAJ [Rho et al. 2006]. For irregular context information, the best solution in these techniques also need to resort to explicit enumeration of base code structure to identify meta-information that need to be accessed. Note that such enumeration increases the coupling between the PCDs and the details of the base code.

1.7. Contributions

This work presents an overview of the Ptolemy language [Rajan and Leavens 2008], which adds quantified, typed events to II languages, producing a language that has many of the advantages of both II and AO languages, but suffers from none of the limitations described above.

Ptolemy declares named event types independently from the modules that announce or handle these events. These event types provide an interface that completely decouples subject and observer modules. An event type p also declares the types of information communicated between announcements of events of type p and handler methods. Events are explicitly announced using **`announce`** expressions. Announce expressions enclose a body expression, which can be replaced by a handler, providing expressiveness akin to **`around`** advice in AO languages. Event type names can also be used in quantification, which simplifies binding and avoids coupling observers with subjects.

Key differences between Ptolemy and II languages are thus:

— separating event type declarations from the modules that announce events,
— the ability to treat an expression's execution as an event,
— the ability to override that execution, and
— quantification by the use of event type names.

Key differences between Ptolemy and AO languages are:

— events are explicitly announced, but quantification over them does not require enumeration unlike techniques such as Open Modules [Aldrich 2005],
— an arbitrary expression can be identified as an event (unlike Setpoint [Altman et al. 2005]) without exacerbating the fragility of PCDs (unlike LogicAJ [Rho et al. 2006]),
— events can communicate an arbitrary set of reflective information to handlers without coupling handlers to program details (cf. [Kellens et al. 2006]), and
— PCDs can use declared event types for quantification.

The benefit of Ptolemy's new features over II languages is that the separation of event type declarations allows further decoupling, and that the ability to replace events completely is more powerful. The benefit over AO languages is that handler methods (advice) can uniformly access reflective information from the context of events without breaking encapsulation of the code that announces events (the "base code"). Event types also permit further decoupling over AO languages, since PCDs are decoupled from the base code.

These benefits make Ptolemy an interesting point in the design space between II and AO languages. Since event announcement is explicit, announcing modules are not completely "oblivious" to the presence of handlers, and hence by some definitions [Filman and Friedman 2000] Ptolemy is not aspect-oriented. However, this lack of obliviousness is not fatal for investigating its utility as a language design, and indeed highlights the advantages and disadvantages of obliviousness, as we will explain Sections 5.2 and 6.

In summary, this work makes the following contributions. It presents:

— a language design with simple and flexible event model;
— a precise operational semantics and type system for the language's novel constructs;
— an implementation of the language as an extension of Eclipse's Java compiler;
— a detailed analysis of our approach and the closely related ideas; and,
— a performance evaluation of Ptolemy's generated code.

2. PTOLEMY'S DESIGN

Ptolemy (Claudius Ptolemaeus), fl. 2d cent. A.D.,
celebrated Greco-Egyptian mathematician, astronomer, and geographer.

In this section, we describe Ptolemy's design. Its use of quantified, typed events extends II languages with ideas from AO languages. Ptolemy features new mechanisms for declaring event types and events. It is inspired by II languages such as Rapide [Luckham and Vera 1995] and AO languages such as AspectJ [Kiczales et al. 2001]. It also incorporates some ideas from Eos [Rajan and Sullivan 2003a; 2005b; 2009] and Caesar [Mezini and Ostermann 2003]. As a small, core language, its technical presentation shares much in common with MiniMAO$_1$ [Clifton 2005; Clifton and Leavens 2006]. The object-oriented part of Ptolemy has classes, objects, inheritance, and subtyping, but it does not have **`super`**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods. The novel features of Ptolemy are found in its event model and type system. In the syntax these novel features are: event type declarations (**`event`**), and announce expressions (**`announce`**).

Like Eos [Rajan and Sullivan 2003a; 2003b; 2005b; 2009; Rajan 2007b], Ptolemy does not have special syntax for "aspects" or "advice". Instead it has the capability to replace all events in a specified set (a pointcut) with a call to a handler method. Each handler takes an event closure as its first argument. An *event closure* contains code needed to run the remaining applicable handlers and the original event's code. An event closure is run by an **`invoke`** expression.

Like II languages, a class in Ptolemy can register handlers for events. However, unlike II languages, where one has to write an expression for registering a handler with each event in a set, Ptolemy allows a handler to be declaratively registered for a set of events by naming the event type in a *binding* (which is similar to declaring AO "around advice"). At runtime, one can use Ptolemy's **`register`** expression to activate such relationships. The **`register`** expression supplies an *observer instance* (an object) that becomes the receiver in calls to its handler methods that are made when the corresponding events are announced.[2] Singleton "aspects" could be easily added as syntactic sugars.

2.1. Syntax

Ptolemy's syntax is shown in Figure 3 and explained below. A program in Ptolemy consists of a sequence of declarations followed by an expression. The expression can be thought of as the body of a "main" method. In Figure 4 we illustrate the syntax using the example from Section 1.

2.1.1. Declarations. The two top-level declaration forms, classes and event type declarations, may not be nested. A class has exactly one superclass, and may declare several fields, methods, and bindings. Bindings associate a handler method to a set of events described by an event type (p). The binding in Figure 4, line 46 says to run `update` when events of type `FEChange` are announced. Similarly, the binding on line 47 says to run `check` when events of type `MoveUpEvent` are announced.

An event type (**`event`**) declaration has a return type (c), a name (p), and zero or more context variable declarations (*form**). These context declarations specify the types and names of reflective information passed by conforming events. Two examples are given in Figure 4 on lines 2 and 5. In writing examples of event types, as in Figure 4, we show each formal parameter declaration

[2] In AO languages such as Eos [Rajan and Sullivan 2003a] and Caesar [Mezini and Ostermann 2003] register expressions correspond to "deploying aspects."

prog ::= *decl** *e*
decl ::= *c* **event** *p* { *form** }
| **class** *c* **extends** *d* { *field* meth* binding** }
field ::= *c f* **;**
meth ::= *t m* (*form**) { *e* }
t ::= *c* | **thunk** *c*
binding ::= **when** *p* **do** *m*
form ::= *t var*, where *var*≠**this**
e ::= **new** *c* () | *var* | **null** | *e* . *m* (*e**) | *e* . *f*
| *e* . *f* = *e* | **cast** *c e* | *form* = *e*; *e* | *e*; *e*
| **register** (*e*) | **announce** *p* (*e**) { *e* } | **invoke** (*e*)

where
$c, d \in \mathcal{C}$, a set of class names
$p \in \mathcal{P}$, a set of evtype names
$f \in \mathcal{F}$, a set of field names
$m \in \mathcal{M}$, a set of method names
$var \in \{\textbf{this}\} \cup \mathcal{V}$, $\mathcal{V}$ is a set of variable names

Fig. 3. Ptolemy's abstract syntax, based on Clifton's dissertation [Clifton 2005, Figures 3.1, 3.7].

```
1 FElement event FEChange{
2  FElement changedFE;
3 }
4 FElement event MoveUpEvent{
5  FElement targetFE; Number y; Number dy;
6 }
7 class FElement extends Object{}
8 class Point extends FElement{ /* ... */
9  Number x; Number y;
10 FElement setX(Number x) {
11  announce FEChange(this){
12   this.x = x; this
13  }
14 }
15 FElement moveUp(Number dy){
16  announce MoveUpEvent (this,y,dy){
17   this.y = this.y.plus(dy); this
18  }
19 }
20 FElement makeEqual(Point other){
21  announce FEChange(other){
22   other.x = this.x;
23   other.y = this.y; other
24  }
25 }
26 }
27 class Update extends Object{
28  FElement last;
29  Update init(){
30   register(this)
31  }
32  FElement update(thunk FElement next,
33      FElement changedFE){
34   FElement res = invoke(next);
35   this.last = changedFE;
36   Display.update(); res
37  }
38  FElement check(thunk FElement next,
39        FElement targetFE,
40        Number y, Number dy){
41   if (dy.lt(100)){
42    FElement res = invoke(next)
43   };
44   targetFE
45  }
46  when FEChange do update
47  when MoveUpEvent do check
48 }
```

Fig. 4. Drawing Editor in Ptolemy

(*form*) as terminated by a semicolon (**;**). In examples showing the declarations of methods, we use commas to separate each *form*. The intention of the first event type declaration (lines 1–3) is to provide a named abstraction for a set of events, with result type `FElement`, that contribute to an abstract state change in a figure element, such as moving a point. This event type declares only one context variable, `changedFE`, which denotes the `FElement` instance that is being changed. Similarly, the event type `MoveUpEvent` (lines 4–6) declares three context variables, `targetFE`, which denotes the `FElement` instance that is moving up, `y`, the current Cartesian co-ordinate value for that instance, and `dy`, the displacement of the instance.

2.1.2. Quantification with Event Types. Event types serve to quantify over the set of events identified by the programmer using announce expressions with the given name. Two examples appear on lines 46–47 of Figure 4. The first, `FEChange`, denotes events identified with the type `FEChange`. The context exposed by this event type is passed explicitly as actual parameters to the event in the announce expressions that mention that type.

2.1.3. Expressions. Ptolemy is an expression language, thus the syntax for expressions includes several standard object-oriented (OO) expressions [Clifton 2005; Clifton and Leavens 2006; Rajan and Leavens 2007].

There are three new expressions: **`register`**, **`invoke`**, and **`announce`**. The expression **`register`**(e) evaluates e to an object o, registers o by putting it into the program's list of active objects, and returns o. The list of active objects is used in the semantics to track registered objects.

Added Syntax:

$e ::= loc \mid \texttt{under}\ e \mid \texttt{NullPointerException} \mid \texttt{ClassCastException}$
where $loc \in \mathcal{L}$, a set of locations

Domains:

$\Gamma ::= \langle e, J, S, A \rangle$	"Configurations"
$J ::= \nu + J \mid \bullet$	"Stacks"
$\nu ::= \texttt{lexframe}\ \rho\ \Pi \mid \texttt{evframe}\ p\ \rho\ \Pi$	"Frames"
$\rho ::= \{j : v_k\}_{k \in K}$, **where** K is finite, $K \subseteq I$	"Environments"
$v ::= loc \mid \texttt{null}$	"Values"
$S ::= \{loc_k \mapsto sv_k\}_{k \in K}$, **where** K is finite	"Stores"
$sv ::= o \mid pc$	"Storable Values"
$o ::= [c\,.\,F]$	"Object Records"
$F ::= \{f_k \mapsto v_k\}_{k \in K}$, **where** K is finite	"Field Maps"
$pc ::= \texttt{eClosure}(H, \theta)\ (e, \rho, \Pi)$	"Event Closures"
$H ::= h + H \mid \bullet$	"Handler Record Lists"
$h ::= \langle loc, m, \rho' \rangle$	"Handler Records"
$A ::= loc + A \mid \bullet$	"Active (Registered) List"

Evaluation contexts:

$$\mathbb{E} ::= - \mid \mathbb{E}\,.\,m(e\ldots) \mid v\,.\,m(v\ldots\mathbb{E}\,e\ldots) \mid \texttt{cast}\ t\ \mathbb{E} \mid \mathbb{E}\,.\,f \mid \mathbb{E}\,;e \mid \mathbb{E}\,.\,f{=}e \mid v\,.\,f{=}\mathbb{E}$$
$$\mid t\ var{=}\mathbb{E};\ e \mid \mathbb{E};\ e \mid \texttt{register}(\mathbb{E}) \mid \texttt{under}\ \mathbb{E} \mid \texttt{invoke}(\mathbb{E}) \mid \texttt{announce}(v\ldots\mathbb{E}\,e\ldots)\{\ e\ \}$$

Fig. 5. Added syntax, domains, and evaluation contexts used in the semantics, based on [Clifton 2005].

Only objects in this list are capable of advising events. For example lines 29–31 of Figure 4 is a method that, when called, will register the method's receiver (**this**). The expression **invoke** (e) evaluates e, which must denote an event closure, and runs that event closure. This runs the handlers in the event closure or, if there are no handlers, the event closure's original expression.

The expression **announce** p (e^*) $\{e'\}$ announces e' as an event of type p and runs any handlers of registered objects that are applicable to p, using a registered object as the receiver and passing as the first argument an event closure. This event closure contains the rest of the handlers, and a closure for the original expression e' with its lexical environment. In Figure 4 the announcement expression on lines 11–13 has a body consisting of a sequence expression. Notice that the announce expressions binds **this** to `changedFE` by giving **this** as the first parameter. This definition makes the value of **this** available in the variable `changedFE`, which is needed by the context declared for the event type `FEChange`. In this figure, the announcement expression declared on line 21–24 also encloses a sequence expression. As required by the event type, the definition on line 21 of Figure 4 binds the value of `other` to `changedFE` by giving it as the first parameter of announce expression. Thus the first and the second event expressions are given different bindings for the variable `changedFE`, however, code that advises this event type will be able to access this context uniformly using the name `changedFE`.

The II syntax "**announce** p" can be thought of as sugar for "**announce** $p()$ $\{$**null**$\}$." Thus Ptolemy's event announcement is strictly more powerful than that in II languages.

2.2. Operational Semantics of Ptolemy

This section defines a small step operational semantics for Ptolemy. This semantics was prototyped in λProlog, using the Teyjus system [Nadathur and Mitchell 1999]. The semantics is based on Clifton's work [Clifton 2005; Clifton and Leavens 2006; Clifton et al. 2007], which builds on Classic Java [Flatt et al. 1999].

The expression semantics relies on four expressions that are not part of Ptolemy's surface syntax as shown in Figure 5. The loc expression represents locations in the store. The **under** expression is used as a way to mark when the evaluation stack needs popping. The two exceptions record various problems orthogonal to the type system.

Figure 5 also describes the configurations, and the evaluation contexts in the operational semantics, most of which is standard and self-explanatory. A configuration contains an expression (e), a

Evaluation relation: $\hookrightarrow : \Gamma \to \Gamma$

(ANNOUNCE)

$$\frac{\rho = envOf(\nu) \quad \Pi = tenvOf(\nu) \quad (c\ \mathbf{event}\ p\{t_1\ var_1, \ldots, t_n\ var_n\}) \in CT \quad \rho' = \{var_i \mapsto v_i \mid 1 \le i \le n\} \quad \pi = \{var_i : \mathbf{var}\ t_i \mid 1 \le i \le n\} \quad loc \notin dom(S) \quad \pi' = \pi \uplus \{loc : \mathbf{var}\ (\mathbf{thunk}\ c)\} \quad \nu' = \mathbf{evframe}\ p\ \rho'\ \pi' \quad H = hbind(\nu' + \nu + J, S, A) \quad \theta = \mathbf{pcd}\ c, \pi \quad S' = S \oplus (loc \mapsto \mathbf{eClosure}(H, \theta)\ (e, \rho, \Pi))}{\langle \mathbb{E}[\mathbf{announce}\ p\ (v_1, \ldots, v_n)\{e\}], \nu + J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ (\mathbf{invoke}(loc))], \nu' + \nu + J, S', A\rangle}$$

(UNDER)

$$\langle \mathbb{E}[\mathbf{under}\ v], \nu + J, S, A\rangle \hookrightarrow \langle \mathbb{E}[v], J, S, A\rangle$$

(REGISTER)

$$\langle \mathbb{E}[\mathbf{register}(loc)], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[loc], J, S, loc + A\rangle$$

(INVOKE-DONE)

$$\frac{\mathbf{eClosure}(\bullet, \theta)\ (e, \rho, \Pi) = S(loc) \quad \nu = \mathbf{lexframe}\ \rho\ \Pi}{\langle \mathbb{E}[\mathbf{invoke}(loc)], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ e], \nu + J, S, A\rangle}$$

(INVOKE)

$$\frac{\mathbf{eClosure}((\langle loc', m, \rho\rangle + H), \theta)\ (e, \rho', \Pi) = S(loc) \quad [c.F] = S(loc') \quad (c_2, t\ m(t_1 var_1, \ldots, t_n var_n)\{e'\}) = methodBody(c, m) \quad n \ge 1 \quad \rho'' = \{var_i \mapsto v_i \mid 2 \le i \le n, v_i = \rho(var_i)\} \quad loc_1 \notin dom(S) \quad S' = S \oplus (loc_1 \mapsto \mathbf{eClosure}(H, \theta)\ (e, \rho', \Pi)) \quad \rho''' = \rho'' \oplus \{var_1 \mapsto loc_1\} \oplus \{\mathbf{this} \mapsto loc'\} \quad \Pi' = \{var_i : \mathbf{var}\ t_i \mid 1 \le i \le n\} \uplus \{\mathbf{this} : \mathbf{var}\ c_2\} \quad \nu = \mathbf{lexframe}\ \rho'''\ \Pi'}{\langle \mathbb{E}[\mathbf{invoke}(loc)], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ e'], \nu + J, S', A\rangle}$$

Fig. 6. Operational semantics of Event-related Expressions

stack (J), a store (S), and an ordered list of active objects (A). Stacks are an ordered list of frames, each frame recording the static environment (ρ) and some other information. (The type environments Π are only used in the type soundness proof.)

There are two types of stack frame. Lexical frames (**lexframe**) record an environment ρ that maps identifiers to values. Event frames (**evframe**) are similar, but also record the name p of the event type being run. Storable values are objects or event closures. Event closures (**eClosure**) contain an ordered list of handler records (H), a PCD type (θ), an expression (e), an environment (ρ), and a type environment (Π). The type θ and the type environment Π (see Figure 8) are maintained by but not used by the operational semantics; they are only used in the type soundness proof. Each handler record (h) contains the information necessary to call a handler method: the receiver object (loc), a method name (m), and an environment (ρ'). The environment ρ' is used to assemble the method call arguments when the handler method is called. The environment ρ recorded at the top level of the event closure is used to run the expression e when an event closure with an empty list of handler records is used in an **invoke** expression.

The initial configuration for a program with main expression e is as follows.

$$\langle \mathbf{under}\ e, (\mathbf{lexframe}\ \{\}\ \{\}) + \bullet, \{\}, \bullet\rangle\,,$$

This configuration starts evaluation of e in a frame with an empty environment, and with an empty store and empty list of active objects.

The rules for standard OO expressions are omitted here, but are shown in Section A.1.

Figure 6 presents the key rules for Ptolemy. The rules all make implicit use of a fixed (global) list, CT, of the program's declarations. The (ANNOUNCE) rule is central to Ptolemy's semantics, as it starts the running of handler methods. In essence, the rule forms a new frame for running the event, and then looks up (using *mbind*) bindings applicable to the new stack, store, and list of registered (active) objects. The resulting list of handler records (H) is put into an event closure (**eClosure**$(H, \theta)\ (e, \rho', \Pi)$)), which is placed in the store at a fresh location. This event closure will execute the handler methods, if any, before the body of the event expression (e) is evaluated. Since a new (event) frame is pushed on the stack, the **invoke** expression that starts running this closure is placed in an **under** expression. The (UNDER) rule pops the stack when evaluation of its subexpression is finished.

$hbind(J, S, \bullet) = \bullet$
$hbind(J, S, loc + A) = concat(hmatch(CT, J, S, loc), hbind(J, S, A))$
where CT is the program's list of declarations
and $concat(\bullet, H') = H'$
$concat(h + H, H') = h + concat(H, H')$

$hmatch(CT, J, S, loc) = match(H, J, S, loc)$ **where** $S(loc) = [c\,.\,F]$ **and** $bindings(CT, c) = H$

$bindings(CT, c) = binds(CT, CT, c)$
$binds(CT, \bullet, c) = \bullet$
$binds(CT, ((t\ \texttt{event}\ p\{\ldots\}) + CT'), c) = binds(CT, CT', c)$
$binds(CT, ((\texttt{class}\ c\ \texttt{extends}\ c'\ \ldots\ binding_1 \ldots binding_n) + CT'), c)$
$= concat((binding_n + \ldots + binding_1 + \bullet), binds(CT, CT, c'))$

$match(\bullet, J, S, loc) = \bullet$
$match(binding + H, J, S, loc)$
$=$ **if** $mpcd(p, J, S) \neq \bot$
then let $\rho = mpcd(p, J, S)$
in let $\rho' = \{var_i \mapsto \rho(var_i) \mid 1 \leq i \leq n\}$
in$(\langle loc, m, \rho' \rangle + match(H, J, S, loc))$
else $match(H, J, S, loc)$
where $binding = \texttt{when}\ p\ \texttt{do}\ m$ **and** $(c\ \texttt{event}\ p\{t_1\ var_1, \ldots, t_n\ var_n\}) \in CT$

$mpcd(p, (\texttt{evframe}\ p'\ \rho\ \Pi) + J, S) =$ **if** $p \equiv p'$ **then** ρ **else** $\bot$

Fig. 7. Auxiliary functions for matching bindings.

The auxiliary function $hbind$, defined in Figure 7 uses the program's declarations, the stack, store, and the list of active objects to produce a list of handler records that are applicable for the event in the current state. When called by the (ANNOUNCE) rule, the stack passed to it has a new frame on top that represents the current event.

The $hmatch$ function determines, for a particular object loc, what bindings declared in the class of the object referred to by loc are applicable. It looks up the location loc in the store, extracts the class of the object loc refers to, and uses that class to obtain a list of potential bindings. This list is filtered using $match$, which relies on $mpcd$ to match the named event type in binding against a particular event on the stack. Each matching binding generates a handler record, recording the active object (which will act as a receiver when the handler method is called), the handler method's name, and an environment. The environment is obtained by $mpcd$, ultimately from the environment in frames of type **`evframe`**. This environment is also restricted to contain just those mappings that are for names in the declared formals of the binding.

When the named event type matches the given stack and store, $mpcd$ returns an environment, otherwise it returns $\bot$. For named events that match, it returns the environment from the top frame on the stack.

The (REGISTER) rule simply puts the object being activated at the front of the list of active objects. The bindings in this object are thus given control before others already in the list. An object can appear in this list multiple times.

The evaluation of **`invoke`** expressions is done by the two invoke rules. The (INVOKE-DONE) rule handles the case where there are no (more) handler records. It simply runs the event's body expression (e) in the environment (ρ) that was being remembered for it by the event closure.

The (INVOKE) rule handles the case where there are handler records still to be run in the event closure. It makes a call to the active object (referred to by loc) in the first handler record, using the method name and environment stored in that handler record. The active object is the receiver of the method call. The first formal parameter is bound to a newly allocated event closure that would run the rest of the handler records (and the original event's body) if it is used in an **`invoke`** expression.

2.3. Ptolemy's Type System

Type checking uses the type attributes defined in Figure 8. The type checking rules themselves are shown in Figure 9. Standard rules for OO features are omitted from here but shown in Section A.2. The notation $\tau' \preccurlyeq \tau$ means τ' is a subtype of τ. It is the reflexive-transitive closure of the declared subclass relationships [Rajan and Leavens 2007].

$\theta ::=$ OK| OK in c | **var** t | **exp** t | **pcd** τ, π "type attributes"
$\tau ::= c \mid \bot$ "class type exps"
$\pi, \Pi ::= \{I : \theta_I\}_{I \in K}$, "type environments"
where K is finite, $K \subseteq (\mathcal{L} \cup \{\texttt{this}\} \cup \mathcal{V})$

Fig. 8. Type attributes.

As in Clifton's work [Clifton 2005; Clifton and Leavens 2006], the type checking rules are stated using a fixed class table (list of declarations) CT, which can be thought of as an implicit (hidden) inherited attribute. This class table is used implicitly by many of the auxiliary functions. We require that the names declared at the top level of a program are distinct and that the extends relation on classes is acyclic.

(CHECK BINDING)

$$\frac{(c_2, c'\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e\}) = methodBody(c, m) \quad \vdash p : \texttt{pcd}\ c', \pi \quad \textit{isClass}(c') \quad n \geq 1 \quad t_1 = \texttt{thunk}\ c' \quad (\forall i \in \{2..n\} :: \textit{isType}(t_i)) \quad \{var_2 : \texttt{var}\ t_2, \ldots, var_n : \texttt{var}\ t_n\} \subseteq \pi}{\Pi \vdash (\texttt{when}\ p\ \texttt{do}\ m) : \text{OK in } c}$$

(CHECK EVENT TYPE)

$$\frac{\textit{isClass}(c) \quad (\forall i \in \{1..n\} :: \textit{isType}(t_i))}{\vdash c\ \texttt{event}\ p\ \{t_1\ var_1\texttt{;}\ \ldots t_n\ var_n\texttt{;}\} : \text{OK}}$$

(EV ID PCD TYPE)

$$\frac{(c\ \texttt{event}\ p\ \{t_1\ var_1\texttt{;}\ \ldots t_n\ var_n\texttt{;}\}) \in CT \quad \pi = \{var_1 : \texttt{var}\ t_1, \ldots var_n : \texttt{var}\ t_n\}}{\vdash p : \texttt{pcd}\ c, \pi}$$

(UNDER EXP TYPE)

$$\frac{\Pi \vdash e : \texttt{exp}\ t}{\Pi \vdash \texttt{under}\ e : \texttt{exp}\ t}$$

(REGISTER EXP TYPE)

$$\frac{\Pi \vdash e : \texttt{exp}\ c}{\Pi \vdash \texttt{register}(e) : \texttt{exp}\ c}$$

(INVOKE EXP TYPE)

$$\frac{\Pi \vdash e : \texttt{exp}\ (\texttt{thunk}\ c)}{\Pi \vdash \texttt{invoke}(e) : \texttt{exp}\ c}$$

(ANNOUNCE EXP TYPE)

$$\frac{(c\ \texttt{event}\ p\ \{t_1\ var_1\texttt{;}\ \ldots t_n\ var_n\texttt{;}\}) \in CT \quad \Pi \vdash e : \texttt{exp}\ c' \quad c' \preccurlyeq c \quad (\forall i \in \{1..n\} :: \Pi \vdash e_i : \texttt{exp}\ t_i)}{\Pi \vdash \texttt{announce}\ p\ (e_1, \ldots, e_n)\ \{e\} : \texttt{exp}\ c}$$

Auxiliary Functions:

$$\textit{isClass}(t) = (\texttt{class}\ t \ldots) \in CT$$
$$\textit{isThunkType}(t) = (t = \texttt{thunk}\ c \wedge \textit{isClass}(c))$$
$$\textit{isType}(t) = \textit{isClass}(t) \vee \textit{isThunkType}(t)$$

Fig. 9. Type-checking rules for Ptolemy. Rules for standard OO features are omitted.

The type checking of method and binding declarations within class c produces a type of the form OK in c, in which c can be considered an inherited attribute. Thus the rule (CHECK BINDING) works with such an inherited attribute c. It checks consistency between c's method m and the PCD. PCD types contain a return type c' and a type environment π, and all but the first formal parameter of the method m must be names defined in π with a matching type. The first formal parameter must be a thunk type that returns the same type, c', as the result type of the method.

Checking event type declarations involves checking that each type used is declared.

Expressions are type checked in the context of a local type environment Π, which gives the types of the surrounding method's formal parameters and declared local variables. Type checking of **under** and **register** is straightforward.

In an expression of the form **invoke**(e), e must have a type of the form **thunk** c, which ensures that the value of e is an event closure. The type c is the return type of that event closure, and hence the type returned by **event**(e).

In an **announce** expression, the result type of the body expression, c', must be a subtype of the result type c declared by the event type, p. Furthermore, the context bindings at an **announce** expression must provide the context demanded by p.

The proof of soundness of Ptolemy's type system uses a standard preservation and progress argument [Wright and Felleisen 1994]. The details are contained in Section A.3.

3. PTOLEMYJ: A JAVA EXTENSION, COMPILER AND RUNTIME SYSTEM

We have designed and implemented an extension of the Java programming language, PtolemyJ that has quantified, typed events. PtolemyJ's compiler targets the standard Java Virtual Machine (JVM). Its front-end is developed as an extension of the Eclipse JDT [Eclipse Foundation 2004]. A highlight of this implementation is that it implements a modular compilation strategy, i.e. to compile a module in our strategy requires accesss to code for that module and the interfaces of static types, but not their implementation. This is achieved at compile-time in contrast to some of the previous efforts that have aimed to achieve the goal via virtual machine extensions [Rajan 2007a; Dyer and Rajan 2008; 2010]. We start with an extended example in PtolemyJ that illuminates some differences between PtolemyJ and AO languages.

3.1. An Extended Example in PtolemyJ

In this section, we present an extended example in PtolemyJ. The example shown in Figure 10 extends the example from Section 1. A set of classes are added to facilitate storing several figure elements in collections, e.g. as a linked list (`FEList`), as a set (`FESet`), and a hash table (`FEHashtable`). Furthermore, `Counter` implements the policy that whenever an `FElement` is added, the count is incremented.

```
1 FElement event FEAdded {FElement addedFE;}
2
3 class FEList {
4  Node listhead;  /*head of linked list*/
5  FElement add(FElement addedFE) {
6   announce FEAdded (addedFE) {
7    Node temp = listhead;
8    listhead = new Node();
9    listhead.data = addedFE;
10   listhead.next = temp;
11   return addedFE;
12  }
13 }
14 FElement remove(FElement fe) { /*...*/ }
15 boolean contains(FElement fe) { /*...*/ }
16 }
17 class FESet extends FEList { /* ... */
18  FElement add(FElement addedFE) {
19   if(!this.contains(addedFE)) {
20    announce FEAdded (addedFE) {
21     Node temp = listhead;
22     listhead = new Node();
23     listhead.data = addedFE;
24     listhead.next = temp;
25     return addedFE;
26    }
27   } else { return null; }
28  }
29 }
30 class Counter {
31  int count;
32  Counter() {
33   register(this);
34  }
35  FElement increment(thunk FElement next,
36    FElement addedFE) {
37   this.count = this.count + 1;
38   return invoke(next)
39  }
40  when FEAdded do increment
41 }
42
43 Counter u = new Counter();
44 /* ... */
```

Fig. 10. Figure Element Collections in PtolemyJ

The notion of "adding an element" differs among the different types of collection. For example, calling `add` on a `FEList` always extends the list with the given element. However, calling `add` on a `FESet` only inserts the element if it is not already present, as shown on lines 19–27. Therefore, an AO-style syntactic method of selecting events such as "an `FElement` is being added" will need to distinguish which calls will actually add the element. In a language like AspectJ, one could use an

if PCD. A PCD such as **call**(* FESet.add(FElement fe)) && **this**(feset) && **if**(!feset.contains(fe)) would filter out undesired call events.

However, there are two issues with using such an **if** PCD. The first issue is **if** PCDs may be expensive at runtime. Second, such a PCD should only be used if the expression feset.contains(fe) does not have any side-effects. (Side-effects would usually be undesirable when used solely for filtering out undesired events.)

Other possibilities for handling such events include: (1) testing the condition in the handler body and (2) rewriting the code for FESet.add to make the body of the **if** a separate method call. The first has problems that are similar to those described above with using an **if** PCD. Rewriting the code to make a separate method call obscures the code in a way that may not be desirable and may cause maintenance problems, since nothing in the code would indicate why the body of the called method was not used inline. There may also be problems in passing and manipulating local variables appropriately in such a body turned into a method, at least in a language like Java or C# that uses call by value.

Such workarounds are also necessary in more sophisticated AO languages such as LogicAJ [Rho et al. 2006]. These have PCDs that describe code structure, but that does not prevent undesirable exposure of internal implementation details, since the structure of the code is itself such a detail.

By contrast, Ptolemy easily handles this problem without exposing internal details of FESet's add method, since that method simply indicates the occurence of event FEAdded. In essence, Ptolemy's advantage is that it can explicitly announce the body of an **if** as an event. Doing so precisely communicates the event without the problems of using **if** PCDs or extra method calls described above.

3.2. Implementation Considerations

The motivation for the careful design and efficient implementation of PtolemyJ stems from the tension between the design flexibility and performance overhead in reusable components that announce events. For increased reuse components must declare and announce sufficient events to support potential observers. For example, the JButton class in javax.swing would need to announce events to expose, among other things, "mouse motion event", "mouse click event", "key pressed event", etc. Indeed in Java 1.6 SDK, this small class announces around 18 events using the actionListener interface. The actionListener type is a widely used interface within Java's SDK for supporting implicit invocation using GOF observer pattern [Gamma et al. 1995].

There is, however, an inherent (fixed) cost to announcing an event using common idioms for implementing observer pattern using imperative style. At each point in the control flow of a component, where an event is announced, code is generally placed to check for registered observers and to invoke them. This cost grows in proportion to the number of events announced by a component. For example, for each of the 18 events declared and announced by the JButton class this checking code is run. In the JDK as currently implemented, even those clients of the JButton class that do not register with all 18 events pay the price of announcing all these events. Thus, the JDK's performance concerns may put a limit on the number of exposed events to minimize the inherent overhead of implicit invocation, which in turn limits how much design flexibility the JDK can efficiently provide by declaring and announcing events.

PtolemyJ's declarative event announcement and registration and its novel implementation strategy reconciles the performance concerns and the design flexibility using several optimization strategies that significantly decreases the inherent cost of announcing an event. This further promotes improved separation of those concerns that are often modularized using observer pattern-like idioms.

Here we describe the key challenges in PtolemyJ's implementation. In describing compilation, we present source code equivalent to the generated bytecode to ease presentation.

3.3. Code Generation for Event Types

The key objective during code generation for event type declarations (**event**) is to preserve the type information during translation. We achieve this goal by mapping event type declarations into

type representations supported by the JVM. Corresponding to an **event** declaration, an interface of the same name is generated. For each formal declared in the event type, an accessor in that interface is generated that has the same name as the name of the formal. The return type of this accessor is the type of this formal and it takes no arguments. The generated code for the **event** FEChange is shown in Figure 11.

```
1 public interface FEChange{
2         public FElement changedFE ();
3         public FElement _invoke ();
4
5         public interface Handler {
6                 public FElement handler(Thunk next, FEChange ev);
7         }
8
9         public final class Thunk {
10                public static List handlers = null;
11                private static Thunk thunks;
12                public static synchronized void register(Handler h){
13                        if(handlers==null)
14                                handlers = Collections.synchronizedList(new ArrayList());
15                        handlers.add(h);
16                        Iterator i = handlers.iterator();
17                        Thunk newThunks = new Thunk(i);
18                        thunkWriteLock.lock();
19                        try {
20                                thunks = newThunks;
21                        } finally {
22                                thunkWriteLock.unlock();
23                        }
24                }
25
26                public static FElement announce(FEChange ev) {
27                        Thunk thunk;
28                        thunkReadLock.lock();
29                        try {
30                                thunk = thunks;
31                        } finally {
32                                thunkReadLock.unlock();
33                        }
34                        return thunk.handler.handler(thunk.nextThunk, ev);
35                }
36
37                public FElement _invoke(FEChange body){
38                        return handler.handler(nextThunk,body);
39                }
40                private Handler handler;
41                private Thunk nextThunk;
42                private Thunk(Iterator chain) {
43                        handler = (Handler) chain.next();
44                        if(chain.hasNext()) {
45                                nextThunk = new Thunk(chain);
46                                return;
47                        }
48                        nextThunk = null;
49                }
50                private static final ReentrantReadWriteLock thunkLock =
51                                                new ReentrantReadWriteLock();
52                private static final Lock thunkReadLock = thunkLock.readLock();
53                private static final Lock thunkWriteLock = thunkLock.writeLock();
54        }
55 }
```

Fig. 11. Generated code for the event type FEChange in Figure 4

As shown in Figure 4 on lines 1–3, the **event** FEChange declares a formal changedFE of type FElement. Therefore, a method changedFE with return type FElement (line 2 in Figure 11) is generated in the interface corresponding to this event type. Another method _invoke

is generated in this interface (line 3 in Figure 11), which serves to invoke the chain of handlers, when the **`invoke`** expression is evaluated in the handler. If no more handlers remain to be invoked, the original event body is evaluated. The return type of this method is the return type of the event type.

A member interface `Handler` (line 5 in Figure 11) and a member class `Thunk` (lines 7-35 in Figure 11) are also generated in the interface corresponding to the event type declaration. The `Handler` interface serves as the type of all handlers in the program that have bindings naming the **`event`**. This interface provides a method `_invoke` to execute the handler method, when events of type `FEChange` are announced.

The objective of the class `Thunk` is to facilitate the desired control flow when the handlers are registered with event types. There are three main methods in this class: `register` (lines 10-17), `announce` (lines 18–23), and `_invoke` (lines 24). The method `register` is used by the handler objects that name the event type in their binding declarations. The complementary method `deregister` is not shown in the figure, but is very similar. The method `register` initializes the list of handlers, if it is not already initialized, adds the registering object `h` to this list, creates the thunks, and caches them for use during event announcement.

The `Thunk` class uses re-entrant read-write locks provided by the Java concurrency API to synchronize the writer method (`register`) and the reader method (`announce`) for the field `thunks`. It ensures that only one thread per event type can register at a time, whereas multiple threads can simultaneously announce events. It also ensures that the set of handlers invoked do not change during an event announcement. This is ensured by creating a new set of thunks during registration (lines 14-16 in Figure 11). The behavior of event announcement that is using an older set of thunks is unaffected by the registration operation that happens later. Thus, there is no need to synchronize the `_invoke` method. Once all announcements that use the older instances of thunks are finished, those become available for garbage collection.

Two optimization strategies are used in the design of the class `Thunk`. First, as opposed to keeping a global list of registered (active) objects, as in Ptolemy's operational semantics (Section 2.2), a list is maintained per event type. This eliminates the overhead of looking up the list of registered objects during every evaluation of an event statement/expression. Second, to avoid a list iteration during successive handler invocation, we expand the handler list into a linked set of thunks (instances of the `Thunk` class), where each such instance contains a reference to the immediate next thunk. This avoids the need for iterating over the handler lists during event announcement. Furthermore, recognizing that announce operations are more frequent compared register operations, we create and memoize these thunks during registration.

3.4. Code Generation for Event Statements and Expressions

The code generation for **`announce`** expressions relies on creating an inner class to emulate event closures and on skip path generation to further optimize event dispatch.

```
1 FElement setX(Number x) {
2  if(FEChange.Thunk.handlers == null) {  this.x = x; return this;  }
3  else {
4    Point$setX$1$Closure point$setX$1 = new Point$setX$1$Closure(this);
5    FEChange.Thunk.announce(point$setX$1);
6 }}
```

Fig. 12. Code Generated for Announce Expressions

Following transformations are done to the body of an announce expression:

(1) The body of the expression is extracted from its original location and cloned.
(2) A class is generated to represent the event closure (Figure 13). The class is named to prevent name clashing using conventions similar to the Java inner classes.

(3) The original body is moved to the true block of a conditional to create a "skip path", where the closure creation code is omitted (line 2 in Figure 12). This if statement tests whether there are any registered handlers. If not, it simply takes the "skip path".
(4) Instructions are generated in the false block of the if statement to create an instance of the closure class and then invoke the announce method on the thunk for the specified event type with the event closure instance as an argument.

In PtolemyJ's implementation when an announce expression's body is extracted to another class, the body may still reference non-final variables in its original scope. Prohibiting such access would have significantly limited the language design. Therefore, we heap allocate such variables and emulate access to them as follows. Each non-final variable that is defined in the enclosing scope and used in the event's body is passed to the event closure's constructor as argument. A field in the event closure is also declared to store each such variable. Any use of such a variable in the code that follows accesses these heap-allocated values under certain conditions. This strategy is similar to that taken by recent efforts to add closures to the Java programming language [Troníček 2007; Gafter 2007].

```
1 final class Point$setX$1$Closure implements FEChange{
2  FElement changedFE; final Point this$0;
3  public Point$setX$1$Closure(Point this$0, FElement changedFE){
4    this.this$0 = this$0; this.changedFE = changedFE;
5  }
6  public FElement changedFE(){ return this.changedFE; }
7  public FElement _invoke(){ this.this$0.x = x; return this.this$0 }
8 }
```

Fig. 13. The inner (generated) class that emulates event closures

Event closure generation proceeds as follows. The clone of the body of the announce expression is inserted into a synthetic method named `_invoke` in the event closure class. Corresponding to each context variable declared by an event type, an accessor is generator in the closure class (an example is in line 6 in Figure 13). A constructor is generated to intialize the fields to their values received from the scope of the event body. Access to non-final variables and lexically scoped constructs in the scope of the event body is managed as described above.

3.5. Code Generation for Bindings and Handlers

The code generation for bindings and handlers proceeds as follows. First, a synthetic interface, a `Handler` member interface of each named event type, is added to the receiver class of the handler. A copy of the target handler method is then created for each event type named in the binding declaration (if such a copy doesn't exist already). The name of this copy is modified to `_invoke`. For example, the argument list is modified to take two arguments of type `FEChange$Thunk` and `FEChange`. All **`invoke`** expressions in this copy are replaced by a call to the method `_invoke` on the argument `next`. Corresponding to each argument of the original method, a local variable declaration is added that explicitly marshals the value from accessors of `next`. Corresponding to each event type that is named in the binding declaration a registration and a deregistration statements are generated in two unique methods `_ptolemy_register` and `_ptolemy_deregister`.

3.5.1. Code Generation for Register Expressions. The register expression is translated into an invocation statement where the target of the invocation statement is the synthetic method `_ptolemy_register` generated during the translation of binding declarations. The parameter of the **`register`** expression becomes the argument of the invocation expression. A benefit of this translation is that the type-checking rule for **`register`** expression is reduced to standard type checking for method call expression.

3.5.2. Resolving Lexically Scoped Constructs. Java has a number of lexically scoped constructs such as name references (e.g. variable names, method names, type names, and special name **`this`**), addresses (e.g. the target of a **`break`**, **`continue`**, and **`return`** statements). As discussed previously, the body of Ptolemy's **`announce`** expression becomes the body of the **`invoke`** method of a synthetic inner class. If these lexically scoped constructs appear within the body of Ptolemy's **`announce`** expressions they are bound to their original lexical scope.

4. RUNTIME PERFORMANCE OF PTOLEMYJ PROGRAMS

In this section, we study the performance aspects of PtolemyJ programs. Two primary scenarios are considered. First, what is the cost of inserting event infrastructure in applications, irrespective of whether they are used? Second, what are the costs of decoupling components that announce events from those that register with them to receive notification using quantified, event types?

The experiments described here were carried out on an Intel Pentium 4 CPU, 3.73GHz with 2GB memory. The Java virtual machine used to run PtolemyJ and other Java programs was the Sun Java Hotspot Client VM build 1.6.0_04 executed with default options. The operating system used was Linux kernel 2.6 running at the runlevel 1, with minimal services running to minimize noise due to operating system services. The results reported in this section are averaged over 30 runs after stabilized performance of the benchmarks.

4.1. Overhead of Event Infrastructure

An important requirement for decoupling components using implicit invocation is that components must explicitly declare and announce events [Sullivan and Notkin 1992; Garlan and Notkin 1991; Notkin et al. 1993]. For example, components in the Java abstract window toolkit (AWT) such as `Button`, etc must announce events such as `ButtonPressed`, etc, so that other components, e.g. a user interface component, can register with these events to receive notification. The infrastructure for event declaration and announcement must remain present in components (in anticipation) irrespective of whether any other component registers or not. Such event infrastructure has a very small but non-zero cost. These costs are important for libraries that must declare and announce all such events that may be of interest to a majority of clients to be useful. For example, the AWT must announce all important events to support various Java applications. In this subsection, we study that cost for Ptolemy programs. A common idiom for manually supporting events is shown in Figure 14.

```
1 public interface Observer { public abstract void notify(int k); }
2 List<Observer> obsrs = Collections.synchronizedList( new ArrayList<Observer>());
3 for(Observer o : obsrs){ o.notify(k); } /* Announcing an event */
```

Fig. 14. A Common Idiom for Supporting Events in an Object-oriented Language

In this idiom, an interface is declared that is implemented by each observer. Here that interface is called `Observer`, but it could have been `ActionListener` as in the AWT. This interface specifies one method, here `notify`, that is invoked by the components announcing the event. A list of such observers is maintained by the component announcing the event. Before announcement this list is checked to ensure if it is non-empty. If yes, the observers in the list are invoked in order. Note that this is less expressive compared to Ptolemy's events that also allow handlers to override events.

To analyze the underlying cost of declaring PtolemyJ's event types and event statements/expressions for a library, we compare and contrast PtolemyJ's implementation with this common idiom used in most object-oriented applications and frameworks. We use the Java Grande benchmark [Bull et al. 2000] for this evaluation as we were just interested in measuring the relative overhead of event dispatch and it was hard to precisely quantify this in a more comprehensive benchmark such as DeCapo [Blackburn, S. *et al.* 2006].

The Java Grande benchmark provides several applications to evaluate the performance of Java implementations. For our purpose, the method invocation benchmark (`MethodBench`) was suf-

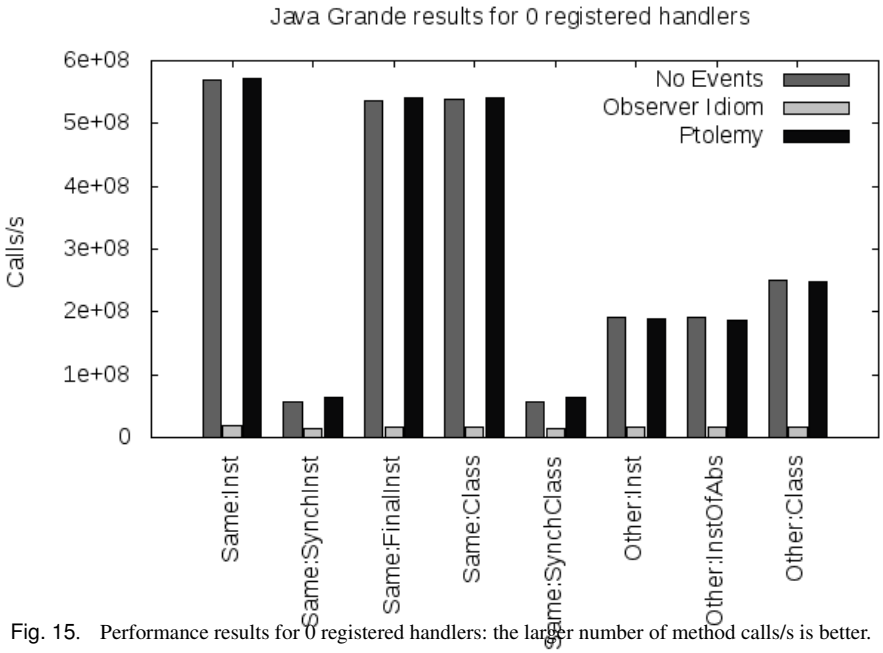

Fig. 15. Performance results for 0 registered handlers: the larger number of method calls/s is better.

ficient. This benchmark measures 8 different invocation strategies in Java applications, non-final, non-synchronized method where the receiver is the currently executing (same) instance, synchronized method on the same instance, final method on the same instance, static method of the same class, synchronized method of the same class, method on other instance, method declared in an abstract class, and static method of other class. These are measured by invoking corresponding methods in the benchmark.

We created two versions of the `MethodBench` suite. In the first version, `ObserverBench`, each method was modified to also announce an event using the observer idiom discussed previously, whereas in the second version, `PtolemyBench`, the body of each such method was enclosed in an **`announce`** expression. For the purpose of results discussed in this subsection, no handlers were registered in either versions. The next subsection analyzes the cost of one or more registered handlers for this benchmark.

The results of running the benchmarks are shown in Figure 15. The X axis in the chart displays the benchmark names and the Y axis displays the method calls per second that were achieved in each case. One can compare two techniques by comparing the number of method calls per second achieved by that technique, the larger the number, the better the technique. For comparison purposes, the results for running the unmodified `MethodBench` are also presented in the figure. For each benchmark the figure first shows the bar for unmodified benchmark followed by the bar for the observer idiom and finally the bar for PtolemyJ's generated code.

The results show that the performance of PtolemyJ's generated code was almost the same as the unmodified benchmark for all cases. Three cases where the PtolemyJ's version shows slight (almost unnoticeable) degradation are Other:Inst, Other:InstOfAbs and Other:Class. In two cases, namely Same:SynchronizedInstance and Same:SynchronizedClass PtolemyJ's version turned out to be slightly better. This was primarily due to the reduced overhead of synchronization required for accessing the list of handlers. The JVM was able to eliminate the extra synchronization. The observer-idiom on the other hand was several order of magnitude slower compared to both the unmodified benchmark and PtolemyJ's version.

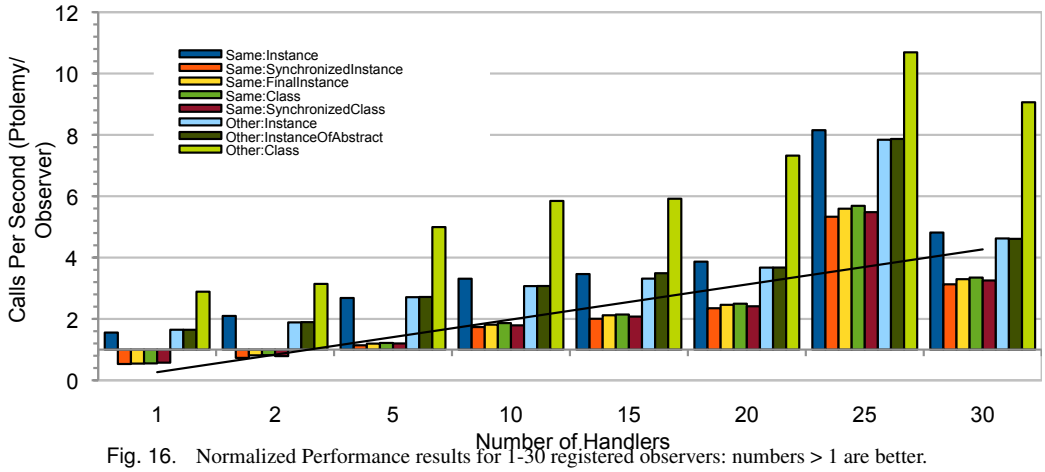

Fig. 16. Normalized Performance results for 1-30 registered observers: numbers > 1 are better.

These results are important from the point of view of the tradeoff between good design and performance. As mentioned previously, an impediment to the use of events for decoupling is the negligible but non-zero overhead that every unused event announcement incurs. By significantly reducing this overhead, PtolemyJ's implementation tilts the balance in favor of extensible design, without sacrificing performance.

4.1.1. Overhead of Event Handler Invocation. Our next objective was to study the overhead of handler invocations and compare the performance of the object-oriented idiom with PtolemyJ's quantified event types as the number of registered handlers increase. We also use the standard Java Grande benchmark [Bull et al. 2000] for this evaluation. Similar to the previous experiment, we created two additional versions of the `MethodBench` program. In the first version, $\texttt{ObserverBench}_n$, each method was modified to also announce an event using the observer idiom, whereas in the second version, $\texttt{PtolemyBench}_n$, the body of each such method was enclosed in an **`announce`** expression. In each case, same number of observers were registered.

The results of runnning the benchmarks are shown in Figure 16. Here the number of registered observers was varied from 1-30. The chart shows the normalized results that is number of method calls per second for PtolemyJ's generated code divided by that seen for for object-oriented idiom. The X axis in the figure represents the number 1 so if the data point is above the axis, PtolemyJ's implementation performed better whereas if it is below the X axis, Observer idiom performed better.

The result show that the performance of PtolemyJ's generated code was almost always better then the hand-written OO-idiom for event announcement. In each event announcement here, PtolemyJ's version was creating a closure, whereas the OO-idiom was not.

These results are important from the point of view of scalability. A trend worth noting is that the slope of the trend line is positive, which shows that as the number of handlers increase the the relative difference between PtolemyJ's generated code and OO idiom widens. This translates to scalability of the technique for a large number of handlers. The code generated by PtolemyJ's compiler is clearly more scalable compared to the OO-idiom.

4.1.2. Summary. In this section, we presented the runtime performance of PtolemyJ programs using Java benchmarks. Our results show that PtolemyJ programs show mostly similar and sometime even better performance compared to common idioms used to emulate support for events in object-oriented languages. The performance improvements mostly come from the optimization techniques implemented by PtolemyJ's compiler. Note that all such optimizations can also be incorporated in the event idioms. Nevertheless PtolemyJ's implementation serves to illustrate that (a) such optimizations can be abstracted behind the language constructs offering programmers the best of both

worlds, and (b) providing declarative constructs will allow the programmers to take advantage of future optimization possibilities without changing their existing codebase.

5. COMPARISONS WITH II AND AO LANGAUGES

The most perfect political community must be amongst those who are in the middle rank. – Aristotle, Politics

In this section we compare Ptolemy with II and AO languages.

5.1. Advance over Implicit Invocation Languages

Consider the II implementation of our drawing editor example (Figure 1). Compared to that implementation, in Ptolemy registration is more automated (see Figure 4), so programmers do not have to write code to register an observer for each separate event.

Ptolemy's registration also better separates concerns, since it does not require naming all classes that announce an event of interest. This is because events are not considered to be attributes of the classes that announce them. Thus, event handlers in Ptolemy need not be coupled with the concrete implementation of these subclasses. Furthermore, naming an event type, such as `FEChange`, in a PCD hides the details of event implementation and allows succinct quantification.

Ptolemy can also replace (or override) code for an event (like AO's "around advice"). Although similar functionality can be emulated in II languages, e.g. by the strategy described in Section 3, Ptolemy's automation significantly eases the programmer's task.

5.2. Advance over AO Languages

Some of the advantages of named event types would also be found in a language like AspectJ 5, which can advise code tagged with various Java 5 annotations. If one only advises code that has certain annotations, then join points become more explicit, and more like the explicitly identified events in Ptolemy. However, Java 5 cannot attach annotations to arbitrary statements or expressions, and in any case such annotations do not solve the problems described in the rest of this section.

5.2.1. Robust Quantified, Typed Events. If instead of lexical PCDs Ptolemy's announce expressions are used and PCDs are written in terms of these event names, innocuous changes in the code that implements the events will not change the meaning of the PCDs. For further analysis of robustness against such changes, let us compare the syntactic PCD **`target`**`(fe) && `**`call`**`(FElement+.set*(..))` with Ptolemy's version in Figure 4, which uses an `announce` expression. The syntactic approach to selecting events provides ease of use, i.e., by just writing a simple regular expression one can select events throughout the program. But this also leads to inadvertent selection of events: **`set`**`*` may select `setParent`, which perhaps does not change the state of a figure element. AO languages with sopisticated matching, based on program structure [Rho et al. 2006] or event history [Stolz and Bodden 2005], have more possibilities for precise description of events, but can still inadvertently select unintended events. Ptolemy's quantified typed events do not have this problem, because they are explicitly announced.

5.2.2. Flexible Quantification. The **`announce`** expression in Ptolemy allows one to label any expression as an event announcement and all such events can be selected by using the event type name in a PCD. Significant flexibility comes from giving developers the ability to decide what expressions constitute events and making them all available for quantification purposes. This largely solves the quantification failure problem [Sullivan et al. 2005]. The events that can be made available to handlers are no longer limited to interface elements, and the implementations of these events are not exposed to handlers. Handlers only rely on event type declarations. In contrast to implicitly announced events in AO languages, **`announce`** expressions allows one to announce any expression as an event.

5.2.3. Flexible Access to Context Information. The third problem that we considered in Section 1 was the difficulty of retrieving context information from a join point. Event types in Ptolemy solve

this problem. To make the reflective information at the event available, a programmer only needs to pass values for in the announce expression corresponding to the formals declared in the event's type. For example, in Figure 4 in the `setX` method an **`announce`** expression passes **`this`** as the actual corresponding to the formal `changedFE`. Note that this flexibility does not introduce additional coupling between events and handlers. Handlers are only aware of the context variable declaration `changedFE` made available by the event type `FEChange` and not of the actual argument expressions used in the announcement.

5.2.4. Uniform Access to Irregular Context Information. AO join point models currently do not provide uniform access to irregular contextual information. But Ptolemy's **`announce`** expressions allow uniform access to such context information. For example, in Figure 4, the announce expression in the `setX` method and in the `makeEqual` method are given different bindings for the context variable `changedFE`, yet the handler `update` is able to access this context information uniformly using the formal parameter `changedFE`. The implementation details are also hidden from PCDs, which can uniformly access the context provided at the event (e.g., using the formal parameter name `changedFE`).

5.2.5. Concern and Obliviousness. Both AO languages and Ptolemy have advantages for certain programming tasks. Consider first whether the concern needs to affect the code in which the events happen — the *base code* in AO terminology. "Spectator" concerns, like tracing, do not affect the base code's state [Clifton and Leavens 2002; Clifton et al. 2007; Dantas and Walker 2006; Xu et al. 2004]. Since spectators do not affect reasoning about the base code, explicit announcements in the base code give little benefit. Hence the determining factor is whether PCDs are easier to write lexically (in the AO style) or using explicitly named events (as in Ptolemy). For syntactically unrelated pieces of code, e.g., the locations of the event `FEAdded` in Figure 10, explicit announcement is more convenient. However, if the events occur in sections of the base code that are syntactically related (by a naming convention, placement in a common package, etc.), then lexical PCDs are preferable.

Besides the availability of uniform context (as described above) another property that affects how easy it is to write lexical PCDs is whether events in the base code are explicit at a module's interface, e.g., calls or executions of a public method. As pointed out by Aldrich [Aldrich 2005] internal events should not be implicitly exported, hence explicit announcement should be used for such events to force negotiation about the commitments involved in having spectators rely on these events.

"Assistants" (i.e., non-spectators [Clifton and Leavens 2002]) have handlers that affect the base code's state. Hence events handled by assistants are important for reasoning about the base code's state. With implicit announcement it is difficult to see these events and take them into account during reasoning. Furthermore, conclusions drawn about the base code will change depending on which assistants are added to the program. Thus we believe that events that are of concern to assistants should always be explicitly announced.

In conclusion, implicit announcement—obliviousness—is useful for spectator concerns when it is easy to write lexical PCDs. In all other cases, Ptolemy's explicit event announcement and its event model are better.

6. COMPARATIVE ANALYSIS WITH RELATED WORK

"There is no other royal path which leads to geometry," said Euclid to Ptolemy I. [Proclus]

In this section, we compare Ptolemy with other mechanisms that address similar problems in AO language design. The other mechanisms we selected for analysis include Aspect-Aware Interfaces (AAIs) [Kiczales and Mezini 2005], Open Modules (OMs) [Aldrich 2005], and Crosscut Programming Interfaces (XPIs) [Sullivan et al. 2005; Sullivan et al. 2011][W. G. Griswold et al. 2006]. The next section summarizes these ideas.

6.1. Overview of Related Ideas

Aspect Aware Interfaces (AAIs) [Kiczales and Mezini 2005] show dependencies between code and handlers. The whole program's configuration, which contains all classes and bindings (including PCDs) is first used to compute dependencies between events and handlers (called the "global step" [Kiczales and Mezini 2005]). The result of this global step is similar in some ways to code in Ptolemy, since one can look at an AAI and see where events may occur that will call handlers, and what handlers may be called for such events. However, whenever the program's bindings are changed, the global step must be repeated and an entirely new set of implicitly announced events might be handled, causing new dependencies. Ptolemy's event expressions do not declare what handlers are applicable for the event they explicitly announce, but the use of explicit announcement ensures that changing a program's bindings will not advise other (previously unanticipated) program points. AAIs also give no help with the problems discussed in Section 1.

Aldrich's Open Modules (OMs) proposal [Aldrich 2005] is closely related to this work and has similar advantages. Like our work, OMs also allows a class developer to explicitly expose the sets of events that are announced. The implementations of these events remain hidden from PCDs and handlers. As a result, the impact of code changes within the class on PCDs is reduced. However, in OMs each explicitly exposed PCD has to be enumerated when binding handlers to sets of events (i.e., when writing advice). By contrast, Ptolemy's event types provide significantly simpler quantification. In Ptolemy, instead of enumerating the events of interest, one can use the event types for more convenient non-syntactic quantification to select join points. As with OMs, a programmer using Ptolemy's event types must systematically modify modules in a system that a given concern crosscuts to expose events that are to be advised, by using **`announce`** expressions. For example, the module *Point* in Figure 4 had to be modified to expose events of type `FEChange`. However, unlike OMs, once modules have incorporated such **`announce`** expressions, no awkward enumeration of explicitly exposed join points is necessary for quantification. Instead, one simply uses the event type `FEChange` in a PCD. Furthermore, in Ptolemy one can expose events that are internal to a module, such as the bodies of **`if`** statements (Figure 10, lines 17–20), which is not possible in OMs (without restructuring the code).

Sullivan *et al.* [Sullivan et al. 2005] proposed a methodology for aspect-oriented design based on design rules. The key idea is to establish a design rule interface that serves to decouple the base design and the aspect design. These design rules govern exposure of execution phenomena as join points, how they are exposed through the join point model of the given language, and constraints on behavior across join points (e.g. *provides* and *requires* conditions [W. G. Griswold et al. 2006]). These design rule interfaces were later called crosscut programming interface (XPI) by Griswold *et al.* [W. G. Griswold et al. 2006]. XPIs prescribe rules for join point exposure, but do not provide a compliance mechanism. Griswold *et al.* have shown that at least some design rules can be enforced automatically. In Ptolemy, enforcing design rules is equivalent to type checking of programs, because event types automatically provide the interfaces needed to decouple different modules.

6.2. Criteria and Analysis Results for Detailed Comparisons

This section discusses more detail about our criteria and the analysis results, which enable a detailed comparison with related work. These comparisons are summarized in Figure 17. The rest of this section presents our analysis in detail.

6.2.1. Abstraction, Information Hiding. The first criterion is whether the approach supports abstraction. All four approaches support abstraction. AAIs abstract the advice that is being executed at the join point, while providing information about the advising structures in a specific system deployment scenario. Their automatically computed abstraction is useful for the developer of the base code in hiding the details of the aspects that may come to depend on the base code. OMs abstract the join point implementation by providing an explicitly declared pointcut as part of the module description. Their abstraction is useful for the aspect code and hides the details of the base code. XPIs provide an abstraction for a set of join points to the aspects, and an abstraction for the possi-

Criteria	Description	AAIs	OMs	XPIs	Ptolemy
Abstraction	Supports abstraction?	Yes	Yes	Yes	Yes
Aspect/Base Il	Is information hiding supported for aspect / base?	Aspect	Base	Aspect + Bas	Aspect + Base
Reasoning	What is the granularity of reasoning?	Join point	Module	XPI's Scope	Expression
Configuration	Requires complete system configuration?	Yes	No	No	No
Decoupling	Decouples aspects from base code?	No	Yes	Yes	Yes
Locality	Are interface definitions textually localized?	No	No	Yes	Yes
Stability	How stable against code changes?	Low	High	Medium	High
Pattern	Allows pattern-based quantification?	Yes	in modu	in XPI's scop	No
Type	Allows quantification based on type hierarchy?	No	No	No	Yes
Scope	What is the scope of the interface?	Program	Module	User defined	User defined
Scope control	Has fine-grained control over scope?	No	No	No	Yes
Adaptation	Requires base code adaption / refactoring?	No	Yes	Yes	Yes
Oblivious	Is it purely oblivious?	No	No	No	No
Lexical hints	Provides lexical hints in a module?	Yes	Yes	No	Yes

Fig. 17. Results of comparative analysis

ble cumulative behavior of all advice constructs to the base program through their requires/provides clauses. Ptolemy provides an abstraction for a set of events to the handlers. It also provide a two-way abstraction for all context information exchanged between a subject and the handler.

6.2.2. Modular Reasoning and the Role of the System Configuration. All four approaches support different mechanisms for modular reasoning. AAIs are different from OMs, XPIs and Ptolemy in that they require that dependencies between base code and aspects be computed before modular reasoning can begin. This may preclude reasoning about a module until all aspects and classes are known. OMs are geared towards supporting reasoning about a change inside a module without knowing about all aspects and classes present in the system. By ensuring that no aspects come to depend upon the changeable implementation details the need to pre-compute all base-aspect dependencies is eliminated. XPIs are geared towards supporting reasoning about a change inside a scope. Ptolemy allows reasoning at the expression level; in particular, only event expressions require any special treatment compared with OO programs.

6.2.3. Locality. This criterion evaluates whether the AO interface definitions are textually localized. AAIs are computed for each point where advice might apply, and thus are not localized. OMs are similar in that the interface of each module explicitly specifies the join points exposed by that module. In XPIs, the AO interface definitions are localized as an abstract aspect. In Ptolemy the event expressions are not localized but the type definition that serves as an interface to the handlers is localized.

6.2.4. Pattern-based Quantification, Scope, and Scope Control Mechanisms. AAIs, OMs and XPIs all support pattern-based quantification. The difference lies in the scope of application of the pattern-based quantification techniques. The scope in the case of AAIs is generally the entire program, but can be limited to specific regions using lexical pointcut expressions such as **`within`** and **`withincode`**. In OMs, they are applicable to inside a module only if used to declare explicitly exposed pointcut and to the entire program if used to select interface elements of modules. XPIs have an explicit scope component that can serve to limit the effect of pattern-based quantification, which in turn is implemented using the **`within`** and **`withincode`** PCDs. In Ptolemy, one can only select program execution events that are declaratively identified. A much finer-grained scope control is available in the case of Ptolemy. In other approaches scope control depends on the language's expressiveness.

6.2.5. Base Code Adaptation and Obliviousness. Obliviousness is a widely accepted tenet for aspect-oriented software development [Filman and Friedman 2000]. In an oblivious AO process, the designers and developers of base code need not be aware of, anticipate or design code to be advised by aspects. This criterion, although attractive, has been questioned by many [Aldrich 2005; Dantas and Walker 2006; W. G. Griswold et al. 2006; Kiczales and Mezini 2005; Steimann 2006; Sullivan et al. 2005]. All four approaches limit the notion of obliviousness to some extent. In Ptolemy adapting base code is necessary.

7. OTHER RELATED IDEAS

advertise, annunciate, broadcast, declare, proclaim, promulgate, publish
– entry for "announce" in Roget's II[Roget 1995]

In some AO langauges quantification is not based on pattern matching of lexical names. For example, in LogicAJ [Rho et al. 2006] and similar languages such as LogicAJ2, Sally [Hanenberg and Unland 2003], quantification is based on program structures, in languages that support trace-based pointcuts [Douence et al. 2001], quantification is based on program traces. As mentioned before, such languages, although significantly more expressive compared to the AspectJ-like languages that quantify based on names, also exhibit fragile pointcut problem. Compared to this entire class of such AO languages, which quantify based on pattern matching, Ptolemy's quantified event types further decouple event handlers and the code that signals events and encapsulates the details of the signaller's code. However, upfront effort is required in Ptolemy to announce events.

Explicitly labeling methods for use in quantification is not a new idea and has appeared previously in SetPoint [Altman et al. 2005] and Model-based Pointcuts [Kellens et al. 2006]. In SetPoint explicitly placed annotations are used for quantification. In Model-based Pointcuts, explictly created models, which express the relationship between names in the model and the program's structure, are used for quantification. Compared to these approaches, the novelty of Ptolemy lies in: allowing arbitrary expressions to be announced as events, in providing explicitly announced events with types, in formalizing the language's sound, static type system, and in providing access to the context of event announcements. Compared to model-based pointcuts, our technique does not require a model construction step. Furthermore, keeping such model consistent with the code can be challenging.

Steimann and Pawlitzki [Steimann et al. 2010] and Hoffman and Eugster [Hoffman and Eugster 2007] have independently advanced ideas that are very similar. Steimann and Pawlitzki's language has event types and explicitly announced events that contain arbitrary statements. Their event types are similar to Ptolemy's. Their language is a modification of AspectJ, and has both implicit (AO style) and explicit announcement of events, whereas Ptolemy only has explicit announcement. Their language is also somewhat similar to Open Modules in that the event types that are exported by a class must be declared by that class. In Hoffman and Eugster's model aspects can declare scoped joinpoint types, that can be used in the base code to mark expressions in a manner similar to Ptolemy's announce expressions. Joinpoint types are part of the aspect and couple base code with the aspect. In contrast, Ptolemy's event types are declared as standalone interfaces to enhance reuse. Both Steimann and Pawlitzki and Hoffman and Eugster have a prototype implementation, but do not formally present their language's semantics or type system. Furthermore, context passing in both their language is limited to the reflective information made available by a typical AspectJ advice, whereas in Ptolemy arbitrary context information can be passed that gives significant flexibility as discussed in Section 5.2.

Delegates in .NET languages such as C# and Java's `EventObject` class are also related to our approach. They are type-safe mechanism for implementing call back functions that can also be used to abstract event declaration code; however, these mechanisms do not provide the quantification feature of Ptolemy's PCDs.

Another related area is mediator-based design styles [Sullivan and Notkin 1992]. In this design style modules tell mediators about event declarations and announcements. Other modules can register with mediators to have their methods invoked by event announcements. An invocation relation is thus created without introducing name dependencies. In our approach, event types play the role of mediators. However, in Ptolemy, one can also use event types for quantification, which simplifies registration and binding. By contrast, in mediator based designs a developer has to resort to explicit and possibly error-prone enumerations to register handlers with events.

Consider a language with closures and the ability to reflectively get the run time context of a statement or expression. In such language, one could achieve the same effect as Ptolemy's quantified event types by declaring classes to represent events, announcing events by creating a closure after reflectively accessing the event body's run time context and then looping over a set of registered

handler methods, passing each a closure (that it could invoke). Ptolemy provides three advantages over this emulation:

— **Static typechecking** of bindings, which ensures that PCDs only associate handlers with events that provide the necessary context.
— A considerable amount of **automation**. Ptolemy's `register`, `event`, and `invoke` expressions hide the details of registration, announcement, and running handlers. Furthermore PCDs provide quantification, which is not easy to emulate.
— **Improved compiler optimizations**. Since Ptolemy controls the details of how registration, announcement, and running handlers are implemented, there is more potential for optimization then when these features are emulated.

8. FUTURE WORK AND CONCLUSIONS

Onward and upward. — Abraham Lincoln

We designed Ptolemy to be a small core language, in order to clearly communicate its novel ability to announce arbitrary expressions as events and its use of quantified, event types, and in order to avoid complications in its theory. However, this means that many practical and useful extensions such as subtyping of event types, cflow PCDs, and use of disjunction in PCDs had to be omitted from the language. We have already extended Ptolemy's operational semantics to include control flow ("cflow") PCDs [Rajan and Leavens 2007] and our implementation allows disjunction in PCDs. These extensions are not discussed in this paper.

The most important future work in the area of Ptolemy's semantics is investigating the possible advantages of Ptolemy's design towards program reasoning and verification. It would also be interesting to combine Ptolemy's type system with an effect system, to limit the potential side effects of handler methods [Clifton 2005; Clifton et al. 2007; Long et al. 2010]. This might allow more efficient reasoning. One could also imagine combining specifications of handler methods into code at `event` expressions, thus allowing verification of code that uses event types to be more efficient and maintainable than directly reasoning about the compiled code's semantics. In general, a detailed investigation of specification and verification issues for Ptolemy would be very interesting and is the subject of some of our current work [Fernando et al. 2012; Bagherzadeh et al. 2011; Bagherzadeh et al. 2013; Bagherzadeh et al. 2015].

In conclusion, the main contribution of this work is the design of a language, Ptolemy, with quantified, typed events. In addition to a precise operational semantics and formal definition of Ptolemy's type system, we have carefully examined how Ptolemy fits in the design space of languages that promote separation of concerns. The main new feature of Ptolemy is event types, which contain information about the names and types of exposed context. In Ptolemy's new event model, events are explicitly announced by event expressions, which declaratively identify the type of event being announced. These event types are used in PCDs to associate handlers with sets of events. Such PCDs are robust against code changes, since they are only affected by changes to event expressions. The event types used in PCDs make it easier for handlers to uniformly access reflective information about the events without breaking encapsulation. Ptolemy has been implemented as a compiler, and its implementation is available for free download [Rajan et al. 2010]. This implementation has also been used in studies about effectiveness of crosscutting interfaces [Dyer et al. 2012; 2013].

Ptolemy's ability to announce any expression as an event, which permits one to expose internal states in a principled way, promises real value. For example, this would help the integration of components when hidden internal states and transitions must be accessed in order to achieve proper composition.

Ptolemy is an open-source project and its compiler and tools are available at http://ptolemyj.sourceforge.net under Mozilla Public License (versio 1.1).

REFERENCES

ALDRICH, J. 2005. Open Modules: Modular reasoning about advice. In *ECOOP*. 144–168.

ALTMAN, R., CYMENT, A., AND KICILLOF, N. 2005. On the need for Setpoints. In *European Interactive Workshop on Aspects in Software*.

BAGHERZADEH, M., DYER, R., FERNANDO, R. D., SANCHEZ, J., AND RAJAN, H. 2015. Modular reasoning in the presence of event subtyping. In *Modularity'15: 14th International Conference on Modularity*.

BAGHERZADEH, M., RAJAN, H., AND DARVISH, A. 2013. On exceptions, events and observer chains. In *AOSD '13: 12th International Conference on Aspect-Oriented Software Development*.

BAGHERZADEH, M., RAJAN, H., LEAVENS, G. T., AND MOONEY, S. 2011. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *AOSD '11: 10th International Conference on Aspect-Oriented Software Development*.

BLACKBURN, S. *et al.*. 2006. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06*. 169–190.

BULL, J. M., SMITH, L. A., WESTHEAD, M. D., HENTY, D. S., AND DAVEY, R. A. 2000. A benchmark suite for high performance Java. *Concurrency: Practice and Experience 12,* 6, 375–388.

CLIFTON, C. 2005. A design discipline and language features for modular reasoning in aspect-oriented programs. Tech. Rep. 05-15, Iowa State University. Jul.

CLIFTON, C. AND LEAVENS, G. 2002. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL*. 33–44.

CLIFTON, C. AND LEAVENS, G. T. 2006. MiniMAO$_1$: Investigating the semantics of proceed. *Science of Computer Programming 63,* 3, 321–374.

CLIFTON, C., LEAVENS, G. T., AND NOBLE, J. 2007. MAO: Ownership and effects for more effective reasoning about aspects. In *ECOOP*. 451–475.

DANTAS, D. S. AND WALKER, D. 2006. Harmless advice. In *POPL 06*. 383–396.

DOUENCE, R., BOTLAN, D. L., NOYï£¡ï£¡, J., AND Sï£¡ï£¡DHOLT, M. 2004. Trace-based aspects. In *In Aspect-oriented Software Development*. Addison-Wesley, 141–150.

DOUENCE, R., MOTELET, O., AND SÜDHOLT, M. 2001. A formal definition of crosscuts. In *REFLECTION '01*. 170–186.

DYER, R. AND RAJAN, H. 2008. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*. ACM, New York, NY, USA.

DYER, R. AND RAJAN, H. 2010. Supporting dynamic aspect-oriented features. *ACM Transactions on Software Engineering and Methodology (TOSEM) 20,* 2.

DYER, R., RAJAN, H., AND CAI, Y. 2012. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *AOSD '12: 11th International Conference on Aspect-Oriented Software Development*.

DYER, R., RAJAN, H., AND CAI, Y. 2013. Language features for software evolution and aspect-oriented interfaces: An exploratory study. *Transactions on Aspect-Oriented Software Development (TAOSD): Special issue, best papers of AOSD 2012 10*, 148–183.

ECLIPSE FOUNDATION. 2004. Eclipse website. http://www.eclipse.org/.

EICHBERG, M., MEZINI, M., AND OSTERMANN, K. 2004. Pointcuts as functional queries. In *APLAS*. 366–381.

FERNANDO, R., DYER, R., AND RAJAN, H. 2012. Event type polymorhphism. In *FOAL '12: Workshop on Foundations of Aspect-Oriented Languages workshop*.

FILMAN, R. E. AND FRIEDMAN, D. P. 2000. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA '00)*.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1999. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*. Springer-Verlag, Chapter 7, 241–269.

GAFTER, N. 2007. Thoughts about the future of the Java programming language: a definition of closures. http://gafter.blogspot.com/2007/01/definition-of-closures.html.

GAMMA, E. AND EGGENSCHWILER, T. 1998. Jhotdraw: a java gui framework for technical and structured graphics. http://sourceforge.net/projects/jhotdraw.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing Co., Inc.

GARLAN, D. AND NOTKIN, D. 1991. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91*. 31–44.

HANENBERG, S. AND UNLAND, R. 2003. Parametric introductions. In *AOSD*. 80–89.

HARBULOT, B. AND GURD, J. R. 2006. A join point for loops in AspectJ. In *AOSD 06*. 63–74.

HOFFMAN, K. AND EUGSTER, P. 2007. Bridging Java and AspectJ through explicit join points. In *PPPJ '07*. 63–72.

KELLENS, A., MENS, K., BRICHAU, J., AND GYBELS, K. 2006. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP*. 501 – 525.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 327–353.

KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP 97*. Finland, 220–242.

KICZALES, G. AND MEZINI, M. 2005. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP*. 195–213.

LONG, Y., MOONEY, S. L., SONDAG, T., AND RAJAN, H. 2010. Implicit invocation meets safe, implicit concurrency. In *GPCE*. ACM, 63–72.

LUCKHAM, D. C. AND VERA, J. 1995. An event-based architecture definition language. *IEEE Trans. Softw. Eng. 21,* 9, 717–734.

MEZINI, M. AND OSTERMANN, K. 2003. Conquering aspects with Caesar. In *AOSD*. 90–99.

NADATHUR, G. AND MITCHELL, D. J. 1999. System description: Teyjus - a compiler and abstract machine based implementation of lambda-prolog. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*. 287–291.

NOTKIN, D., GARLAN, D., GRISWOLD, W. G., AND SULLIVAN, K. J. 1993. Adding implicit invocation to languages: Three approaches. In *JSSST International Symposium on Object Technologies for Advanced Software*. 489–510.

PROCLUS. Commentary on Euclid's Elements. Book ii, chapter iv. From Bartlett's Familiar Quotations, 10th edition, 1919.

RAJAN, H. 2007a. A case for explicit join point models for aspect-oriented intermediate languages. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*. ACM Press, New York, NY, USA, 4.

RAJAN, H. 2007b. Design pattern implementations in eos. In *PLoP '07, Conference on Pattern Languages of Programs*.

RAJAN, H. AND LEAVENS, G. T. 2007. Quantified, typed events for improved separation of concerns. Tech. Rep. 07-14c, Iowa State University, Dept. of Computer Sc. Oct.

RAJAN, H. AND LEAVENS, G. T. 2008. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*.

RAJAN, H., LEAVENS, G. T., MOONEY, S., DYER, R., AND BAGHERZADEH, M. 2010. Ptolemy website. http://ptolemyj.sourceforge.net.

RAJAN, H. AND SULLIVAN, K. 2003a. Eos: instance-level aspects for integrated system design. In *the European software engineering conference and international symposium on Foundations of software engineering (ESEC/FSE)*. 297–306.

RAJAN, H. AND SULLIVAN, K. 2003b. Need for instance level aspect language with rich pointcut language. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies Workshop colocated with AOSD 2003*.

RAJAN, H. AND SULLIVAN, K. 2005a. Aspect language features for concern coverage profiling. In *the international conference on Aspect-oriented software development (AOSD)*. 181–191.

RAJAN, H. AND SULLIVAN, K. J. 2005b. Classpects: unifying aspect- and object-oriented language design. In *the international conference on Software engineering (ICSE)*. 59–68.

RAJAN, H. AND SULLIVAN, K. J. 2009. Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology (TOSEM) 19,* 1 (August).

RHO, T., KNIESL, G., AND APPELTAUER, M. 2006. Fine-grained generic aspects. In *FOAL*.

ROGET, Ed. 1995. *Roget's II: The New Thesaurus*, Third Edition ed. Houghton Mifflin Company.

STEIMANN, F. 2006. The paradoxical success of aspect-oriented programming. In *OOPSLA '06*. 481–497.

STEIMANN, F., PAWLITZKI, T., APEL, S., AND KASTNER, C. 2010. Types and modularity for implicit invocation with implicit announcement. *ACM TOSEM 20,* 1.

STOERZER, M. AND GRAF, J. 2005. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM*. 653–656.

STOLZ, V. AND BODDEN, E. 2005. Temporal assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV '05)*.

SULLIVAN, K. J., GRISWOLD, W. G., RAJAN, H., SONG, Y., CAI, Y., SHONLE, M., AND TEWARI, N. 2011. Modular aspect-oriented design with xpis. *ACM Transactions on Software Engineering and Methodology (TOSEM) 20,* 2.

SULLIVAN, K. J., GRISWOLD, W. G., SONG, Y., CAI, Y., SHONLE, M., TEWARI, N., AND RAJAN, H. 2005. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*. 166–175.

SULLIVAN, K. J. AND NOTKIN, D. 1992. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology 1,* 3 (July), 229–68.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*.

TOURWÉ, T., BRICHAU, J., AND GYBELS, K. 2003. On the existence of the AOSD-evolution paradox. In *SPLAT*.

TRONÍČEK, Z. 2007. Closures: Implementation issues. http://tronicek.blogspot.com/2007/12/closures-closure-is-form-of-anonymous_28.html.

W. G. GRISWOLD ET AL. 2006. Modular software design with crosscutting interfaces. *IEEE Software*.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1 (Nov), 38–94.

XU, J., RAJAN, H., AND SULLIVAN, K. 2004. Understanding aspects via implicit invocation. In *Automated Software Engineering (ASE)*. IEEE Computer Society Press, 332–335.

A. APPENDIX: SEMANTICS, TYPE RULES, AND SOUNDNESS PROOF

In this section, we present the omitted rules for semantics and type checking rules for object-oriented expressions and the proof of type soundness.

A.1. Omitted Semantics of Object-oriented Expressions

Figure 18 presents the evaluation rules for standard OO expressions in Ptolemy.

Evaluation relation: $\hookrightarrow : \Gamma \to \Gamma$

(NEW)
$$\frac{loc \notin dom(S) \qquad S' = S \oplus (loc \mapsto [c.\{f \mapsto \texttt{null} \mid f \in dom(fieldsOf(c))\}])}{\langle \mathbb{E}[\texttt{new}\ c()], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[loc], J, S', A\rangle}$$

(CAST)
$$\frac{[c'.F] = S(loc) \qquad c' \preccurlyeq c}{\langle \mathbb{E}[\texttt{cast}\ c\ loc], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[loc], J, S, A\rangle}$$

(VAR)
$$\frac{\rho = envOf(\nu) \qquad v = \rho(var)}{\langle \mathbb{E}[var], \nu + J, S, A\rangle \hookrightarrow \langle \mathbb{E}[v], \nu + J, S, A\rangle}$$

(GET)
$$\frac{[c.F] = S(loc) \qquad v = F(f)}{\langle \mathbb{E}[loc.f], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[v], J, S, A\rangle}$$

(SET)
$$\frac{[c.F] = S(loc) \qquad S' = S \oplus (loc \mapsto [c.F \oplus (f \mapsto v)])}{\langle \mathbb{E}[loc.f = v], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[v], J, S', A\rangle}$$

(SKIP)
$$\langle \mathbb{E}[v\,;\, e], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[e], J, S, A\rangle$$

(DEF)
$$\frac{\rho = envOf(\nu) \qquad \rho' = \rho \oplus (var \mapsto v) \qquad \Pi = tenvOf(\nu) \qquad \Pi' = \Pi \uplus \{var : \texttt{var}\ t\} \qquad \nu' = \texttt{lexframe}\ \rho'\ \Pi'}{\langle \mathbb{E}[t\ var = v\,;\, e], \nu + J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\texttt{under}\ e], \nu' + \nu + J, S, A\rangle}$$

(CALL)
$$\frac{\begin{array}{c}[c.F] = S(loc) \\ (c_2, t\ m(t_1 var_1, \ldots, t_n var_n)\{e\}) = methodBody(c, m) \qquad \rho = \{var_i \mapsto v_i \mid 1 \le i \le n\} \oplus (\texttt{this} \mapsto loc) \\ \Pi = \{var_i : \texttt{var}\ t_i \mid 1 \le i \le n\} \uplus \{\texttt{this} : \texttt{var}\ c_2\} \qquad \nu = \texttt{lexframe}\ \rho\ \Pi\end{array}}{\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\texttt{under}\ e], \nu + J, S, A\rangle}$$

Fig. 18. Operational semantics of OO expressions in Ptolemy, based on [Clifton 2005].

The (NEW) rule says that the store is updated to map a fresh location to an object of the given class that has each of its fields set to null. This rule (and others) uses $\oplus$ as an overriding operator for finite functions. That is, if $S' = S \oplus (loc \mapsto v)$, then $S'(loc') = v$ if $loc' = loc$ and otherwise $S'(loc') = S(loc')$. The *fieldsOf* function uses the class table to determine the list of field declarations for a given class (and its superclasses), considered as a mapping from field names to their types.

In the (VAR) rule, $envOf(\nu)$ returns the environment from the current frame ν, ignoring any other information in ν.

$$\begin{array}{rcl} envOf(\texttt{lexframe}\ \rho\ \Pi) & = & \rho \\ envOf(\texttt{evframe}\ p\ \rho\ \Pi) & = & \rho \end{array}$$

Thus the (VAR) rule says that the value of a variable, including **this**, is simply looked up in the environment of the current frame. The (CALL) rule implements dynamic dispatch by looking up the method m starting from the dynamic class (c) of the receiver object (loc), looking in superclasses if necessary, using the auxiliary function $methodBody$ (not shown here). The body is executed in a **lexframe** with an environment that binds the methods formals, including **this**, to the actual parameters. Since methods do not nest, and since expressions access object fields by starting from an explicit object there is no other context available to a method.

Note that **under** e is used in the resulting configuration for the (CALL) rule. This expression is used whenever a new frame is pushed on the stack, to record that the stack should be popped when the evaluation of e is finished. As described previously, the (UNDER) rule pops the stack when

evaluation of its subexpression is finished. The (GET) and (SET) rules are standard. The value of a field assignment is the value being assigned.

The (CAST) rule simply checks that the dynamic class of the object is a subtype of the type given in the expression. The (NCAST) rule allows **null** to be cast to any type.

The (DEF) rule allows for local definitions. It is similar to **let** in other languages, but with a more C++ and Java-like syntax. It simply binds the variable given to the value in an extended environment. Since a new frame is pushed on the stack, the body, e, is evaluated inside an "under" expression, which pops the stack when e is finished. The (SKIP) rule for sequence expressions is similar, but no new frame is needed.

A.2. Omitted Type Checking Rules for Object-oriented Expressions

$$\text{(NEW EXP TYPE)}\ \frac{\mathit{isClass}(c)}{\Pi \vdash \texttt{new}\ c\,()\ :\ \texttt{exp}\ c} \qquad \text{(CAST EXP TYPE)}\ \frac{\mathit{isClass}(c)}{\Pi \vdash \texttt{cast}\ c\ e\ :\ \texttt{exp}\ c} \qquad \text{(NULL EXP TYPE)}\ \frac{\mathit{isClass}(c)}{\Pi \vdash \texttt{null}\ :\ \texttt{exp}\ c}$$

$$\text{(GET EXP TYPE)}\ \frac{\Pi \vdash e : \texttt{exp}\ c \qquad \mathit{fieldsOf}(c)(f) = t}{\Pi \vdash e.f\ :\ \texttt{exp}\ t} \qquad \text{(SET EXP TYPE)}\ \frac{\Pi \vdash e : \texttt{exp}\ c \qquad \mathit{fieldsOf}(c)(f) = t \qquad \Pi \vdash e' : \texttt{exp}\ t' \qquad t' \preccurlyeq t}{\Pi \vdash e.f{=}e' : \texttt{exp}\ t'}$$

$$\text{(DEF EXP TYPE)}\ \frac{\mathit{isType}(t) \qquad \Pi \vdash e_1 : \texttt{exp}\ t_1 \qquad t_1 \preccurlyeq t \qquad \Pi' = \Pi \uplus \{\mathit{var} : \texttt{var}\ t\} \qquad \Pi' \vdash e_2 : \texttt{exp}\ t_2}{\Pi \vdash t\ \mathit{var}{=}\ e_1 ; e_2 : \texttt{exp}\ t_2}$$

$$\text{(SEQ EXP TYPE)}\ \frac{\Pi \vdash e_1 : \texttt{exp}\ t_1 \qquad \Pi \vdash e_2 : \texttt{exp}\ t_2}{\Pi \vdash e_1 ; e_2 : \texttt{exp}\ t_2} \qquad \text{(VAR EXP TYPE)}\ \frac{(\mathit{var} : \texttt{var}\ t) \in \Pi}{\Pi \vdash \mathit{var} : \texttt{exp}\ t}$$

$$\text{(CALL EXP TYPE)}\ \frac{\Pi \vdash e : \texttt{exp}\ c \qquad (c_2, t\ m\ (t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\ \{e\}) = \mathit{methodBody}(c, m) \qquad c \preccurlyeq c_2 \qquad \Pi \vdash e_1 : \texttt{exp}\ t_1\ \ldots\ \Pi \vdash e_n : \texttt{exp}\ t_n}{\Pi \vdash e.m(e_1, \ldots, e_n) : \texttt{exp}\ t}$$

$$\text{(NP EXCEPTION EXP TYPE)}\ \Pi \vdash \texttt{NullPointerException} : \texttt{exp}\ \bot \qquad \text{(CC EXCEPTION EXP TYPE)}\ \Pi \vdash \texttt{ClassCastException} : \texttt{exp}\ \bot$$

Fig. 19. Type-checking rules for OO features.

Figure 19 presents the type checking rules for standard OO expressions in Ptolemy. See Clifton's thesis [Clifton 2005] for details on these straightforward rules.

A.3. Type Soundness

The proof of soundness of Ptolemy's type system uses a standard preservation and progress argument [Wright and Felleisen 1994]. The details are adapted from Clifton's work [Clifton 2005; Clifton and Leavens 2006], which in turn follows Flatt *et al.*'s work [Flatt et al. 1999]. Throughout this section we assume a fixed, well-typed program with a fixed class table.

The key idea in the proof of the subject-reduction theorem is the preservation of consistency between the type environment and the stack and store. This notion is built on the following notion of a (non-null) location having a particular type in the store. This involves fields holding values of their declared types and consistency of the type information in an event closure.

Definition A.1 (loc *has type* t *in* S). Let loc be a location, t be a type, and S be a store. Then loc *has type* t *in* S if and only if one of the following holds:

(a) $isClass(t)$ and for some c and F: (i) $S(loc) = [c.F]$, (ii) $c \preccurlyeq t$, (iii) $dom(F) = dom(fieldsOf(c))$, (iv) $rng(F) \subseteq (dom(S) \cup \{\texttt{null}\})$, and (v) for all $f \in dom(F)$, if $F(f) = loc'$, $fieldsOf(c)(f) = u$, and $S(loc') = [c'.F']$, then $c' \preccurlyeq u$

(b) $isThunkType(t)$, $t = \textbf{thunk}\ c$, and for some H, π, e, ρ, Π, and c' such that all the following hold: (i) $S(loc) = \textbf{eClosure}(H, \textbf{pcd}\ c, \pi)(e, \rho, \Pi)$, (ii) $\Pi \vdash e : \textbf{exp}\ c'$, (iii) $c' \preccurlyeq c$, (iv) for each $var_i \in dom(\Pi)$, if $(var_i : \textbf{var}\ t_i) \in \Pi$ then $\rho(var_i)$ has type t_i in S, (v) for each $loc_i \in dom(\Pi)$, if $(loc_i : \textbf{var}\ t_i) \in \Pi$ then loc_i has type t_i in S, and (vi) for each handler record h in H, h has type $\textbf{pcd}\ c, \pi$ in S.

The last notion used in the above definition is defined as follows.

Definition A.2 (h *has type* $\textbf{pcd}\ c, \pi$ *in* S). Let h be the handler record $\langle loc, m, \rho \rangle$, let c be a class name, π a type environment, and S a store. Then h *has type* $\textbf{pcd}\ c, \pi$ *in* S if and only if for some c', F, c_2, t', $n > 1$, var_i, t_i and e: $S(loc) = [c'.F]$, $methodBody(c', m) = (c_2, c\ m(t_1 var_1, \ldots, t_n var_n)\{e\})$, $dom(\rho) = dom(\pi) = \{var_2, \ldots, var_n\}$, $t_1 = \textbf{thunk}\ c$, and for each $i \in \{2, \ldots, n\}$, $(var_i : \textbf{var}\ t_i) \in \pi$ and $\rho(var_i)$ has type t_i in S.

The key definition of consistency is thus as follows. In the definition, $tenvOf(\nu)$ is the type environment of a frame ν, and $envOf(\nu)$ returns ν's environment. Notice that the type environment (Π) can have some locations in its domain; these are needed to enable the typing of location expressions. (Location expressions are used in the semantics of **new** expressions, for example.)

Definition A.3 (*Environment-Stack-Store Consistent*). Let Π be a type environment, J a stack, and S a store. Then Π *is consistent with* (J, S), written $\Pi \approx (J, S)$, if and only if either $J = \bullet$ or $J = \nu + J'$ and all the following hold:

(1) $\Pi = tenvOf(\nu)$,
(2) if $\rho = envOf(\nu)$, then for all $(var : \textbf{var}\ t) \in \Pi$, $var \in dom(\rho)$ and $\rho(var)$ has type t in S, and
(3) for all $(loc : \textbf{var}\ t) \in \Pi$, $loc \in dom(S)$ and loc has type t in S.

The subject-reduction theorem, as usual, says that evaluation steps preserve both types and consistency. The key idea that makes preservation of consistency easy to prove is the use of type information buried in frames and event closures. This type information is maintained by the operational semantics, but not used by it. Maintenance of this type information occurs each time the stack changes (since the type environment must match that of the top stack frame), and each time a chain expression is created.

THEOREM A.4 (SUBJECT-REDUCTION). *Let e be an expression, J a stack, S a store, and A an active object list. Let Π be a type environment and t a type. If $\Pi \approx (J, S)$, $\Pi \vdash e : \textbf{exp}\ t$, and $\langle e, J, S, A \rangle \hookrightarrow \langle e', J', S', A' \rangle$, then there is some Π' and t' such that $\Pi' \vdash e' : \textbf{exp}\ t'$, $t' \preccurlyeq t$ and $\Pi' \approx (J', S')$.*

Proof Sketch: The proof is by cases on the definition of $\hookrightarrow$ (see Figure 6 and Figure 18). Assume $\Pi \approx (J, S)$, $\Pi \vdash e : \textbf{exp}\ t$, and $\langle e, J, S, A \rangle \hookrightarrow \langle e', J', S', A' \rangle$.

The OO cases (rules (NEW), (GET), (SET), (CAST), (NCAST), and (SKIP)) are all straightforward, and can be proved by simple adaptions of Clifton's proofs for MiniMAO$_0$ [Clifton 2005, Section 3.1.4]. The result for the exception cases all follow directly from the use of $\textbf{exp}\ \bot$ as their type and the fact that the stack in the resulting configuration is empty.

The (CALL) rule is different from Clifton's MiniMAO$_0$, and thus must be handled in detail. This case is also a good illustration of how the type information in the configurations is preserved. Suppose $e = loc.m(v_1, \ldots, v_n)$. From the hypotheses of the (CALL) rule we have that: $[c.F] = S(loc)$, $(c_2, t''\ m(t_1 var_1, \ldots, t_n var_n)\{e''\}) = methodBody(c, m)$, $\rho = \{var_i \mapsto v_i \mid 1 \leq i \leq n\} \oplus (\textbf{this} \mapsto loc)$, $\Pi'' = \{var_i : \textbf{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{\textbf{this} : \textbf{var}\ c_2\}$, and $\nu = \textbf{lexframe}\ \rho\ \Pi''$. So in this case, $e' = \textbf{under}\ e''$, $J' = \nu + J$, and $S' = S$. Since the program is assumed to be well-typed, by the (CHECK PROGRAM) typing rule, all its declarations type check, and so by the (CHECK CLASS) rule, the class c_2 where m is defined type checks, and so by the (CHECK METHOD) rule, the method m type checks in class c_2. Thus by the hypotheses of the (CHECK METHOD) rule we can choose Π' to be Π'' and t' to be t''. That rule also gives us that

$\Pi' \vdash e'' : \mathbf{exp}\, t'$ and $t' \preccurlyeq t$. To prove $\Pi' \approx (\nu + J, S')$ we use definition A.3. The first condition holds by construction, since the type environment of ν is equal to Π'', which is our Π'. The second condition holds because for each var_i, if $\rho(var_i) = loc_i \neq \mathbf{null}$, then the loc_i has type t_i in S, because for e to be well-typed, it must be that $\Pi \vdash v_i : \mathbf{exp}\, t_i$ (due to the hypotheses of the (CALL EXP TYPE) rule), and by assumption $\Pi \approx (J, S)$. The third condition is vacuous in this case.

The case for the (DEF) rule is similar, and is also similar to Clifton's (SEQ) case.

Preservation is trivial for the (REGISTER) case, since we can choose $t' = t$. Consistency is also trivial in this case, since the rule makes no changes to the stack or store.

For the (ANNOUNCE) rule, suppose $e = \mathbf{announce}\ p\ \{e''\}$. From the conclusion of this rule it must be that $J = \nu + J''$ for some ν and J''. From the hypotheses of the (ANNOUNCE) rule we have that: $\rho = envOf(\nu)$, $\Pi'' = tenvOf(\nu)$, $(c\ \mathbf{event}\ p\{t_1\ var_1, \ldots, t_n\ var_n\}) \in CT$, $\rho' = \{var_i \mapsto v_i \mid \rho(var_i) = v_i\}$, $\pi = \{var_i : \mathbf{var}\, t_i \mid 1 \leq i \leq n\}$, $loc \notin dom(S)$, $\pi' = \pi \uplus \{loc : \mathbf{var}\ (\mathbf{thunk}\ c)\}$, $\nu' = \mathbf{evframe}\ p\ \rho'\ \pi'$, $H = hbind(\nu' + \nu + J, S, A)$, $\theta = \mathbf{pcd}\, c, \pi$, and $S' = S \oplus (loc \mapsto \mathbf{eClosure}(H, \theta)\ (e, \rho, \Pi''))$. So in this case $e' = \mathbf{under}\ (\mathbf{invoke}(loc))$ and $J' = \nu' + \nu + J''$. To preserve consistency, we must choose $\Pi' = \pi'$ since that is the type environment of frame ν'. Since by hypothesis $\Pi \vdash e : \mathbf{exp}\, t$, by the (EVENT EXP TYPE) rule, we have that $c = t$, and so we can choose $t' = c$, and thus $t' \preccurlyeq t$. With these choices $\Pi' \vdash e' : \mathbf{exp}\, t'$, using the type rules (UNDER EXP TYPE) and (INVOKE EXP TYPE), since $\Pi' \vdash loc : \mathbf{exp}\ (\mathbf{thunk}\ c)$ by construction. To prove $\Pi' = \pi' \approx (\nu' + \nu + J'', S')$ we use definition A.3. The first condition holds by construction. The second condition holds because the variables in the domain of Π' are a subset of those in the domain of Π, π and ρ' are constructed with matching domains, and the only change to S' from S is the addition of loc. The third condition holds because the only location in the domain of Π' is loc, which has type $\mathbf{thunk}\ c$ in S' by construction.

For the (INVOKE-DONE) rule, suppose $e = \mathbf{invoke}(loc)$. From the hypothesis of this rule $\mathbf{eClosure}(\bullet, \theta)\ (e'', \rho'', \Pi'') = S(loc)$ and $\nu = \mathbf{lexframe}\ \rho''\ \Pi''$. So in this case we have $e' = \mathbf{under}\ e''$, $J' = \nu + J$, and $S' = S$. To preserve consistency, we choose $\Pi' = \Pi''$, which is the type environment originally used to type check e''. Since by hypothesis, $\Pi \vdash \mathbf{invoke}(loc) : \mathbf{exp}\, t$, from the (INVOKE EXP TYPE) rule, we have that $\Pi \vdash loc : \mathbf{exp}\ (\mathbf{thunk}\ t)$. By hypothesis, we know that $\Pi \approx (J, S)$, and hence by definition loc has type $\mathbf{thunk}\ t$ in S. Thus $\Pi'' \vdash e'' : \mathbf{exp}\ c''$, where $c'' \preccurlyeq t$. So we choose $t' = c''$, which makes $t' \preccurlyeq t$. It follows directly from the (UNDER EXP TYPE) that $\Pi'' \vdash \mathbf{under}\ e'' : \mathbf{exp}\ c''$. To prove $\Pi' = \Pi'' \approx (\nu + J, S')$ we again use definition A.3. The first condition holds by construction. The second and third conditions hold because of the hypothesis that $\Pi \approx (J, S)$, hence loc has type $\mathbf{thunk}\ t$ in $S = S'$, and thus these conditions hold by parts (b)(iv) and (b)(v) in definition A.1.

For the (INVOKE) rule, suppose $e = \mathbf{invoke}(loc)$. From the hypothesis of this rule: $\mathbf{eClosure}((\langle loc', m, \rho\rangle + H), \theta)\ (e'', \rho'', \Pi'') = S(loc)$, $[c.F] = S(loc')$, $(c_2, t''\ m(t_1 var_1, \ldots, t_n var_n)\{e'''\}) = methodBody(c, m)$, $n \geq 1$, $\rho_3 = \{var_i \mapsto v_i \mid 2 \leq i \leq n, v_i = \rho(var_i)\}$, $loc_1 \notin dom(S)$, $S' = S \oplus (loc_1 \mapsto \mathbf{eClosure}(H, \theta)\ (e'', \rho'', \Pi''))$, $\rho_4 = \rho_3 \oplus \{var_1 \mapsto loc_1\} \oplus \{\mathbf{this} \mapsto loc'\}$, $\Pi_3 = \{var_i : \mathbf{var}\, t_i \mid 1 \leq i \leq n\} \uplus \{\mathbf{this} : \mathbf{var}\, c_2\}$, and $\nu = \mathbf{lexframe}\ \rho_4\ \Pi_3$. So in this case we have $e' = \mathbf{under}\ e'''$ and $J' = \nu + J$. To preserve consistency, we choose $\Pi' = \Pi_3$. Since by hypothesis, $\Pi \vdash \mathbf{invoke}(loc) : \mathbf{exp}\, t$, from the (INVOKE EXP TYPE) rule, we have that $\Pi \vdash loc : \mathbf{exp}\ (\mathbf{thunk}\ t)$. By hypothesis, we know that $\Pi \approx (J, S)$, and hence by definition loc has type $\mathbf{thunk}\ t$ in S. Thus by definition A.1 (b)(vi), the handler record $\langle loc', m, \rho\rangle$ has type $\theta = \mathbf{pcd}\, t, \pi$ in S. By definition A.2, $t'' = t$, $\pi = \{var_2 : \mathbf{var}\, t_i, \ldots, var_n : \mathbf{var}\, t_n\}$, $t_1 = \mathbf{thunk}\, t$, and for each $i \in \{2, \ldots, n\}$, $\rho(var_i)$ has type t_i in S. Then since the program is assumed to be well-typed, by the (CHECK METHOD) rule, using the hypothesis that the return type of m is t'', we have that $\Pi_3 \vdash e''' : \mathbf{exp}\ t''''$ and $t'''' \preccurlyeq t''$. So we choose $t' = t'' = t$, which makes $t' \preccurlyeq t$. It follows directly from the (UNDER EXP TYPE) that $\Pi_3 \vdash \mathbf{under}\ e''' : \mathbf{exp}\ t'$. To prove $\Pi' = \Pi_3 \approx (\nu + J, S')$ we again use definition A.3. The first condition holds by construction. The second condition holds because: (1) $\rho_4(var_1) = loc_1$ and by construction loc_1 has type $t_1 = \mathbf{thunk}\, t$ in S', (2) by construction for each $i \in \{2, \ldots, n\}$, $\rho_4(var_i) = \rho(var_i)$, and $\rho(var_i)$ has type t_i in S, which holds the same values as S' for these lo-

cations, and (3) $\rho_4(\texttt{this}) = loc'$ which has type c in S and hence in S', and so by definition of $methodBody$, $c \preccurlyeq c_2$, which is the type of **this** in Π_3. The third condition is vacuous in this case. ∎