Quantified, Typed Events for Improved Separation of Concerns

Hridesh Rajan and Gary T. Leavens

TR #07-14d

Keywords: implicit-invocation languages, aspect-oriented programming languages, quantification, pointcut, join point, context exposure, type checking, event types, event expressions.

CR Categories:
D.3.3 [Programming Languages] Language Constructs and Features - Control structures

Copyright (c) 2008, Hridesh Rajan and Gary T. Leavens.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA
Quantified, Typed Events for Improved Separation of Concerns

Hridesh Rajan  
Iowa State University  
hridesh@cs.iastate.edu

Gary T. Leavens  
University of Central Florida*  
leavens@eecs.ucf.edu

Abstract

Implicit invocation and aspect-oriented languages provide related but distinct mechanisms for separation of concerns. Implicit invocation languages have explicitly announced events, which runs registered observer methods. Aspect-oriented languages have implicitly announced events, called “join points,” which run method-like but more powerful advice. A limitation of implicit invocation languages is their inability to refer to a large set of events succinctly. They also lack the expressive power of aspect-oriented advice, and require code to manage event registration and announcement. Aspect-oriented languages also have several limitations, including the potential for fragile dependence on syntactic structure that may hurt maintainability, limits in the set of join points and the reflective contextual information that they make available.

Quantified, typed events solve all these problems. They extend implicit invocation languages with a key idea from aspect-oriented languages: the ability to quantify over events (join points). Programmers declare named event types that contain information about the names and types of event arguments (exposed context). An event type declaratively identifies an expression as an event. This event type can then be used to quantify over all such events. Event types reduce the coupling between the observers and the set of events, and similarly between the advising and advised code.

1. Introduction

The objective of both implicit invocation (II) [13, 34] and aspect-oriented (AO) [11] languages is to improve a software engineer’s ability to separate conceptual concerns. The problem that they address is that all dimensions of design decisions, or concerns, are not amenable to modularization by a single dimension of decomposition. Instead, some concerns cut across the dominant dimension of decomposition. These approaches aim to improve the separation of these types of concerns thereby enhancing modularity.

The key idea in II languages is that modules declare, announce, and register with events. Announcing an event means running all observer methods registered for that event, without explicitly naming them. On the other hand, AO languages such as AspectJ [12] use predicates, called pointcuts, to select events in the program’s execution. These events are called join points. A language’s features for pointcuts and its set of join points form its join point model. Using pointcuts to select join points to be advised, called quantification [9], is one of the main ideas in AO languages. Quantification crucially depends on the language’s join point model.

II Languages have three limitations. First, they require complex event declaration, announcement and registration code scattered across the system [26]. Second, the ability to use around advice, which replaces the code for an event, is not available in II languages, without unnecessarily complex emulation code (to construct closures in languages such as Java and C#). Third, quantification of events is not easy. That is, using an abstraction similar to pointcuts in AOP to refer to a collection of related events is difficult. Instead, a non-trivial strategy such as a subscription registry [20] is needed.

AO languages also have some limitations. These limitations arise because most current join point models use lexical pointcuts. Such pointcuts refer to join points, such as method calls, by named patterns, such as set*, which would name all methods with a name starting with “set”. Except for a few approaches such as SetPoint [2], functional queries [8], etc, the prominent means of quantification are lexical. Lexical pointcuts are fragile [32, 35], exhibit quantification failures [33], and make it unnecessarily hard to uniformly access relevant context at the join point [33] (see Section 2.1-2.4).

This work’s contribution is Ptolemy, a language with quantified, typed events. When an event type is declared, it is given a name, which can be used in quantification. An event type t also declares the types of information communicated between events and observer methods for events of type t. Events are declaratively identified using event expressions that name the event’s type.

Key differences between Ptolemy and II languages are: (i) the ability to treat an expression’s execution as an event, (ii) the ability to override that execution, (iii) abstraction of event registration, announcement code, and (iv) quantification. Key differences between Ptolemy and AO languages like AspectJ [12] are: (i) join points are declaratively identified, (ii) an arbitrary expression can be identified as join point, (iii) they can expose an arbitrary set of reflective information, and (iv) they can be selected using event types. Since one can tell when advice will be run, Ptolemy is not purely oblivious, and hence by some definitions [9] Ptolemy is not aspect-oriented.

The benefit of Ptolemy’s new features over II languages is that observer methods are decoupled from the code that announces events, instead they only name event types. The benefit over AO languages is that advice can uniformly access reflective information about the join point without breaking its encapsulation, thus it is decoupled from the base code structure and the names used.

We describe this model in what follows. Sections 2 and 3 motivate and present our approach and language design. Section 4 illustrates key properties of our language design. Section 5 compares our proposed approach with other similar approaches. We offer some discussion in Section 7. Section 6 compares our approach with related work and Section 8 concludes.

* Much of Leavens’s work was done while he was still at Iowa State.

[Copyright notice will appear here once ‘preprint’ option is removed.]
2. Motivation

In this section, we illustrate the limitations of implicit invocation and aspect-orientation using a simple editor for drawings comprising points, lines, and other such figure elements [12, 19] shown in Figure 1 (ignoring the gray code for now). The listings show a Point and Line class. The Point class uses two integers to store Cartesian coordinates, and provides methods to set these coordinates. The Line class stores two points and provides a method to set the coordinates of the two end points of this line. This method in turn sets the Cartesian co-ordinates by accessing the fields in- 
side the class to keep a list of observers for each event exposed, by extending the setter methods to run the notify method on all observers.

The class Display (not listed) manages the display and provides a method update to keep the state of the figure elements consistent. The greyed part of this figure implements the policy that the display must be updated whenever the abstract state of a FElement changes. This is done by declaring the observer interface, by extending the Point and Line class to keep a list of observers for each event exposed, by extending the setter methods to run the notify method on all observers.

Two limitations of implicit invocation are evident in this example. The event declaration, announcement and registration code (shown in gray) is complex and scattered across the Point and Line classes. Quantification of events requires the registration code to explicitly enumerate events of interest in registerWithPoint and registerWithLine methods.

The listings in Figure 2 shows an alternative AspectJ implementation for the drawing editor discussed before. Notice now that in AO implementation, Point and Line classes are free of any event related code. The third part of the figure accomplishes the modularization of display update policy using the aspect Update. An aspect can select all points that change the abstract state of all figure elements by writing pointcut descriptions (PCDs) such as target(fe) && (call(FElement.set*(..))). This PCD selects all join points that change the state of a FElement and binds fe to the changed FElement. This pointcut expression will select appropriate join points only if all such join points in the program are systematically exposed [36]. At all these join points the advice is run, which updates the display, then runs the original call, using proceed.

AO languages fix the limitations of II languages, but they suffer from four problems, which we explain in the rest of this section.

2.1 Fragile Pointcuts

The first problem is due to use of syntactic predicates as a quantification mechanism [32, 35]. Such predicates are likely to change in the face of base code modifications.

To illustrate, consider a simple refactoring of the class Point in Figure 2 to hide the implementation details by making the fields x and y private. This change requires Line’s developer to use the methods setX and setY as shown below.

Figure 2. Drawing editor’s AO implementation: aspect in gray

in the relevant list, and having the class Update add (register) itself to the lists for events of interest.
This seemingly innocuous change breaks the aspect \texttt{Update}. This aspect will now update the screen three times for every change in the end points of an instance of the class \texttt{Line}. It updates once when the call to \texttt{setP1} occurs, a second time when the call to \texttt{setX} occurs, and a third time when the call to \texttt{setY} occurs. This change should have been encapsulated in the classes \texttt{Point} and \texttt{Line}, but it is propagating to the advising code. This is an obvious maintenance problem, which would be magnified in a real example.

### 2.2 Quantification Failure

The second problem is what Sullivan et al.\cite{33} have called \textit{quantification failure}. In the context of the AO design of the Hypercast system, they observed that many join points that have to be advised cannot be captured by a quantified pointcut descriptor (PCD), instead a separate PCD is required for each join point \cite[pp. 170]{33}. They also observe that many join points of interest are not available as interface elements, but are instead deeply embedded into methods.\footnote{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.}

These join points occur in places such as inside iteration and conditional statements. Exposing such join points as additional language constructs \cite{14, 27} seems to be a solution to the quantification failure. However, these constructs further couple the aspects with the base code and expose the implementation details of the base code, violating encapsulation.

The root of quantification failure lies in existing techniques for join point classification and quantification. These techniques work by classifying events in the program’s execution as different kinds of join points, such as execution, call, field access, etc. We can understand these techniques better by drawing an analogy to untyped set theory. Let \( J \) be the set of all potential join points in a program. The join point classification can be thought of as partitioning \( J \) into disjoint subsets \( J_{kind} \), for each \( kind \) in some set \( KIND \) of different kinds of join points. Some of these subsets may not be available in a given join point model. For example, iteration, conditional, and most expressions are not available in AspectJ’s model.

The limitation of this view of join point classification is that it is fixed by the programming language designer. This contributes to quantification failure, because new kinds of join points cannot be defined by developers. Quantification failure arises mainly because in existing join point models developers cannot specify their own decomposition of the base program. As long as the developer uses an object-oriented decomposition based on classes and methods, current quantification mechanisms work remarkably well and a large set of join points can be selected using succinct pointcut descriptions. However, if a different decomposition is needed to modularize a concern, then language models need explicit enumerations, and pointcut descriptions become verbose and more fragile.

### 2.3 Limited Access to Context Information

The third problem is the difficulty of retrieving context information from a join point. Current aspect languages provide an interface for accessing contextual (or reflective) information about a join point. A fundamental problem is that, in current languages, this interface between the join point and advice is fixed by the language designer. For example, in AspectJ, advice can access contextual information at the join point using pointcuts such as \texttt{this} to access the executing object, \texttt{target} to access the receiver object of a call, \texttt{args} to access the arguments at a join point, etc. Alternatively, one can explicitly marshall this information from a reflection object \texttt{thisJoinPoint}. Unfortunately, this rather limited interface does not satisfy all usage scenarios.

Even the canonical concerns such as logging exhibit these problems. For modularizing the logging functionality in a program, aspect developers need access to the context of the join points that are to be logged. This information is often stored in local variables in the source code surrounding the join point. However, access to local variables is not available in existing join point models.

There are rational reasons for limiting the interface between the advised and the advising code. This interface couples the design of the advised and advising code. The thinner this interface is the lower the coupling will be, resulting in perhaps easier and independent evolution of these two designs \cite{30}. Extending the set of language constructs to include access to more primitives also takes away regularity from the language design \cite{21}, because not all such primitives will be valid for all kinds of join points. As it is, current language constructs for retrieving contextual information are not completely regular; e.g., \texttt{this}, \texttt{target}, and \texttt{args} are not available at all join points in AspectJ.

### 2.4 Uniform Access to Irregular Context Information

The fourth problem with the current join point models is their inability to retrieve the same contextual information from different join points selected in one pointcut description. Advice attached to such a pointcut description needs to access equivalent contextual information at each join point.

To illustrate this problem, consider the listing in Figure 3 extracted from the class \texttt{Point}. This listing shows two method implementations.

```java
public class Point implements FElement { // ... *
    public void setX(int x) { this.x = x; }
    public void makeEqual(Point other) {
        other.x = this.x; other.y = this.y;
    }
}
```

\textbf{Figure 3.} Two methods in \texttt{Point} affect different context.

In this simple case, it is possible to work around this issue by rewriting the PCD as follows.

```java
(target(fe) && (call(FElement.set*(...)))) || (args[fe] && call(FElement.makeEqual(...)))
```

However, such rewriting is undesirable for two reasons. First and most importantly, each such specialization that we apply to the pointcut couples it with the details of the base code. Second, for advice that applies to a large number of such irregular join points, the pointcut expression may become significantly large and thus hard to maintain. We will see a solution to these problems in the next section.

### 3. Ptolemy’s Design

In this section, we describe Ptolemy, a language with quantified, typed extensions that extends implicit invocation (II) languages with...
ideas from aspect-oriented (AO) languages. Ptolemy features new mechanisms for declaring event types and events. Our description includes syntax, examples, semantics, and type checking rules. An example is given in Figure 4.

3.1 Overview

Ptolemy is inspired by II languages such as Rapide [20] and AO languages such as AspectJ [12]. It also incorporates some ideas from Eos [25] and Caesar [22]. As a small, core language, its technical presentation shares much in common with Clifton and Leavens's MiniMAO [3, 5], which itself builds on Classic Java [10] and Featherweight Java [16]. The object-oriented part of Ptolemy follows MiniMAO. While it has classes, objects, inheritance, and subtyping, it does not have super interfaces, exception handling, built-in value types, type provisors, or abstract methods. The novel features of Ptolemy are found in its event model and type system. In the syntax these novel features are: an event type declaration (evtype), and an event expression (event).

Like Eos [25], Ptolemy does not have special syntax for aspects and advice. Instead it has the capability to replace all events in a specified set (a pointcut) with a call to a method. Following II terminology, we call such methods event handlers or simply handlers. Each handler takes an event closure as its first argument. An event closure [25] (or delegate, or delegate chain) contains code needed to run the applicable handlers and the original event expression. An event closure can be run using an invoke expression.

Like II languages a Ptolemy module can register to receive event notifications. This capability is the same as "deployment" in the IA languages Eos and Caesar [22]. However, like II languages, registration makes explicit the receiver instance that will run the handler, and allows instance-level advising features to be easily programmed [26]. Singleton "aspects" that are created at the start of the program and automatically registered can also be easily programmed or added as a syntactic sugar.

3.2 Syntax

Ptolemy's syntax is shown in Figure 5 and explained below. A program consists of a sequence of declarations followed by an expression. The expression can be thought of as the body of a (static, or receiverless) "main" method. We next explain declarations, pointcut descriptions, and expressions.

prog ::= decl* e
decl ::= class : extends { field* meth* binding* }
       | c evtype p | form*
field ::= cf;
meth ::= t m (form*) { e }
t ::= c { thunk;
bind ::= when pcd do m;
form ::= t var, where var this pcd
pcd ::= p cflow(pcd) | pcd && pcd | pcd | pcd

c ::= new c () | var null | e.m(e*) | e.f | c.e.f = e
| cast c e | form = e; e | e;
| register (e) | event p { e } | invoke (e)

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

Figure 5. Ptolemy's Abstract syntax, based on [3, Figure 3.1, 3.7].

3.2.1 Declarations

There are only two declaration forms that may appear at the top level of a Ptolemy program: classes and event type declarations. These may not be nested. A class has exactly one superclass, named in its extends clause. It may declare several fields (field*), methods (meth*), and bindings (binding*). Field declarations are written with a class name, giving the field's type, followed by a field name. Methods also have a C++ or Java-like syntax, although their body is an expression. As in Eos, bindings associate a set of events, described by a pointcut description (PCD), to a method. An example is shown in Figure 4, which contains a binding on line 28. The binding tells Ptolemy to run the method update whenever events of type FEChange are executed.

An event type (evtype) 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 exposed by conforming events. An example is given in Figure 4 on line 18. In writing examples of event types, as in Figure 4, we show each formal parameter declaration (form) as terminated by a semicolon (.). In examples showing the declarations of methods and bindings, we use commas to separate each form.

The intention of this event type declaration is to provide a named abstraction for a set of events, with resultant type FElement, that contribute to an abstract state change in a figure element, such as moving a point, line, etc. This example event type declares only one context variable, changedFE, which denotes the FElement instance that is being changed. An event can only be of this type if: (a) it has the stated result type and (b) it binds the context variable changedFE to some value in its lexical scope, as shown in Figure 4 (lines 5–8).

3.2.2 Quantification: Pointcut Descriptions

The syntax for pointcut descriptions (or PCDs, sometimes called pointcut designators) has one basic form and three recursive forms, corresponding to basic and complex events in II languages. The basic PCD is the named PCD, which denotes the set of events that are identified by the programmer using event expressions with that name. The context exposed by such a named event is
the context available at the event identified by the programmer. An example appears in lines 28–29 of Figure 4, which denotes events identified with the type FEChange.

The cflow, or control flow, PCD is similar to AspectJ’s cflow PCD. It names all programmer-identified events that occur during the execution of the PCD it contains, including those named by that PCD. The context exposed by such a cflow PCD is the context exposed by the underlying PCD. For example cflow (FEChange) includes all events in FEChange, as well as all those that occur during their execution, and it exposes all the context that FEChange exposes. However, unlike AspectJ, in Ptolemy only explicitly identified events that occur in the control flow of FEChange are considered to be events, not all possible events of AspectJ’s predefined event kinds. This change makes clear where advice can run.

As in AspectJ, disjunction (|) of two PCDs gives the union of the sets of events denoted by the two PCDs. The context exposed by the disjunction is the intersection of the context exposed by the two PCDs. However, if an identifier I is bound in both contexts, then I’s value in the exposed context is I’s value from the right hand PCD’s context.

Similarly, the conjunction of two PCDs intersects the set of events denoted by the two PCDs. A conjunction event exposes context that is the union of the context exposed by the two PCDs. Again, if an identifier I is bound in both contexts, then I’s value in the exposed context is I’s value from the right hand PCD’s context.

3.2.3 Expressions

Ptolemy is an expression language. Thus the syntax for expressions includes several standard object-oriented (OO) expressions and also some expressions that are specific to aspects.

The standard OO expressions include object construction (new e()), variable dereference (var, including this), field dereference (e.f), null, cast (cast t e), assignment to a field (e1 = e2), a definition block (t var = e1; e2), and sequencing (e1; e2). Their semantics and typing is fairly standard [3, 5].

There are also three new expressions: register, event, and invoke.

The expression register(e) evaluates e to an object o, registers o by putting it into the list of active objects, and returns o. The list of active objects is used in the semantics to track registered objects. Only objects in this list are capable of advising events. For example line 21 of Figure 4 is a method that, when called, will register the method’s receiver (this).

The expression event p{ e} declares the expression e as an event of type p and runs any handler methods of registered objects (i.e., those in the list of active objects) that are applicable to p. That is, it marks e as the shadow [15] of an event of type p. Note that only (well-formed) expressions can produce events, one may not, describe an an event that contains only part of an expression. The type named, p, must be an event type. This type name: (i) identifies the event for purposes of quantification, much like an annotation would in AspectJ 3, and (ii) is used in type checking.

The expression invoke(e) evaluates e, which must denote an event closure, and runs that event closure. This results in running the first handler method in the chain of applicable handlers in the event closure. If there are no such handler methods, it runs the original expression from the event.

When called from an event, or from invoke, each handler method is called with a registered object as its receiver. The call passes an event closure as the first actual argument to the handler method.

An example demonstrating these features is shown in Figure 4. The event declared on lines 6–8 has a body consisting of the sequence expression on line 7. Notice that the body of the setX method contains a block expression, where the definition on line 5 binds this to changedFE, and then evaluates its body, the event expression. This definition makes the value of this available in the context variable changedFE, which is needed by the event type FEChange. In this figure, the event declared on lines 12–15 encloses the sequence expression on lines 13–14. As required by the event type, the definition on line 11 of Figure 4 makes the value of other available in the context variable changedFE.

Thus the first and the second event expressions are given different bindings for the context variable changedFE, however, code that advises this event type will be able to access this context variable uniformly using the name changedFE.

The evaluation of an expression event first looks for any applicable bindings for objects in the active (registered) list. The handler methods from such applicable bindings are formed into a list, ordered in reverse of the order of object activation, with the most recently registered object’s handlers first. The list is put into an event closure, which also remembers the event expression’s body. Then the first handler, if any, is run; if it invokes, it will run the next handler, or the body expression if there are no more handlers.

This ordering of handlers in the event closure is designed to allow more recently registered objects to control whether previously registered objects have their handlers run, by using invoke (or not). Similarly, within an object’s handlers, subclass and textually later bindings are allowed the same control over superclass and textually earlier bindings. That is, when handler methods from applicable bindings for an object are formed into a list, handlers from that subclasses of that object’s type appear before handlers declared in its superclasses. Furthermore, for bindings declared in the same class, handlers for textually earlier bindings appear after handlers for later bindings.

Consider a Ptolemy program that combines Figure 4 followed by the main expression

```
Update u = new Update().init();
Point p = new Point();
p.setX(new Zero());
```

This main expression creates and registers an Update object, which it names u. It then creates a Point object, and binds it to p. The call to setX binds the formal x to the object representing the number 0, and then runs the body of setX. Since the body contains an event expression, and since there is an active object (u) that contains a binding for that event, that binding’s handler method is run. This method, update, is called with receiver u, an event closure as the first argument, and the value of changedFE as the second argument. The body of update assigns to u’s field last, and runs the event closure (the expression invoke(next)), which executes the body of the event expression (starting at line 7 of Figure 4) in its original environment. The body of the event expression returns the value of this, which, since the environment of the call to setX has been restored, is the value of p. This value is returned as the value of the handler chain, and hence as the result of the method update. In turn, this result, p, is used as the value of the event expression, and hence as the value of the call to setX. Thus the expression in the last line of the main program’s expression, u.last denotes the same object as p.

The grammar only allows one event type to be named in an event expression. However, it is convenient to allow a list of event types as a syntactic sugar. The desugaring to a nest of event expressions is as follows.

```
ev
```

Note, however, that this sugar does not make the events listed occur simultaneously; they instead occur in a definite order.

6
3.3 Operational Semantics

This section defines a small step operational semantics for Ptolemy. This semantics has been implemented in the logic programming language XProlog, using the Teyjus system [23]. This semantics and its description in this section is adapted from Clifton’s work [3, 5, 6], which builds on Classic Java [10]. Following these works, a program’s declarations are simply formed into a fixed list, which is used in the semantics of expressions. The small steps of the operational semantics thus gives a semantics of programs by giving a semantics of expressions.

The expression semantics relies on four expressions that are not part of Ptolemy’s surface syntax. These expressions allow the semantics to record final or intermediate states of the computation, and are shown in Figure 6. 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.

\[ e := \text{loc} \mid \text{under} \mid \text{NullPointerException} \mid \text{ClassCastException} \]

Figure 6. Added syntax for Ptolemy’s operational semantics.

The small steps taken in the semantics transition from one configuration to another. These configurations are described in Figure 7. A configuration contains an expression (\( e \)), a stack (\( S \)), and an ordered list of active objects (\( A \)). Stacks are an ordered list of frames, each frame recording the static environment (\( \rho \)) and some other information. (The type environments \( \Pi \) are only used in the type soundness proof.) There are two types of stack frame. Lexical frames (\( \text{lexframe} \)) record an environment \( \rho \) that maps identifiers to values. Event frames (\( \text{evframe} \)) are similar, but also record the name of the event type being run. A value is a location or \( \text{null} \). Stores are maps from locations to storable values. Storable values are either objects or event closures. Objects have a class and also a map from field names to values. Event closures (\( \text{eclosure} \)) contain an ordered list of handler records (\( H \)), a PCD type (\( \theta \)), an expression (\( e \)), an environment (\( \rho \)), and a type environment (\( \Pi \)). The type \( \theta \) and the type environment \( \Pi \) (see Figure 12) are not used by the operational semantics, but only in the type soundness proof. Each handler record (\( h \)) contains the information necessary to call a handler method: a value that will be the receiver object of the method call (\( \text{loc} \)), a method name (\( m \)), and an environment (\( \rho_h \)). The environment \( \rho_h \) is used to assemble the method call arguments when the handler method is called. The environment \( \rho \) recorded at the top level of the event closure is used to run the expression \( e \) when an event closure with no handler records is used in an `invoke` expression.

As is usual [37] the semantics is presented as a set of evaluation contexts \( \mathcal{E} \) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context. This two-part presentation avoids the need for writing out standard recursive rules and has the advantage of more clearly presenting the order of evaluation.

Figure 7 defines evaluation contexts and hence the order of evaluation for Ptolemy. The language uses a strict leftmost, innermost evaluation policy, which thus uses call-by-value. The initial configuration for a program with main expression \( e \) is \( \langle \text{under} \ e, \text{lexframe} \rangle \), which starts evaluation of \( e \) in a frame with an empty environment, and with an empty store and empty list of active objects.

Figure 8 defines the operational semantics of Ptolemy. In these rules all of the hypotheses are really side conditions and side definitions for use in the rule.

Domains:

\[
egin{align*}
\Gamma ::= & \langle \text{e}, \text{J}, \text{S}, \text{A} \rangle & \text{ "Configurations"} \\
J ::= & \nu + \text{J} \mid \bullet & \text{ "Stacks"} \\
\nu ::= & \text{lexframe} \rho \Pi & \text{ "Lexical"} \\
\text{evframe} \rho \Pi & \text{ "Event execution"} \\
\rho ::= & \{j:v_k\}_{k\in K}, & \text{ "Environments"} \\
\text{where} \ K \text{ is finite}, K \subseteq I \\
v ::= & \text{loc} \mid \text{null} & \text{ "Values"} \\
S ::= & \{lok \leadsto sv_k\}_{k \in K}, & \text{ "Stores"} \\
\text{where} \ K \text{ is finite} \\
sv ::= & a[pc] & \text{ "Storable Values"} \\
o ::= & [e,F] & \text{ "Object Records"} \\
F ::= & \{f_\nu \mid v_k\}_{k \in K}, & \text{ "Field Maps"} \\
\text{where} \ K \text{ is finite} \\
pc ::= & \text{eclosure}(H, \theta) (e, \rho, \Pi) & \text{ "Event Closures"} \\
H ::= & h + H \mid \bullet & \text{ "Handler Record Lists"} \\
h ::= & \langle \text{loc}, m, \rho \rangle & \text{ "Handler Records"} \\
A ::= & \text{loc} + A \mid \bullet & \text{ "Active (Registered) List"} \\
\end{align*}
\]

Figure 7. Domains used in the semantics, based on [3].

Evaluation contexts:

\[
\begin{align*}
\mathcal{E} ::= & - \mid \mathcal{E}.m(e... \mid v.m(v... \mathcal{E}e... \mid \text{cast} t \mathcal{E} \\
& | \mathcal{E}.f(e \mid \mathcal{E}.f = e \mid v.\mathcal{F} = e \mid \mathcal{E} \mid \mathcal{E} \mid \mathcal{E} \mid \mathcal{E} \mid \mathcal{E}) \\
& | \text{register}(\mathcal{E}) \mid \under \mathcal{E} \mid \text{invoke}(\mathcal{E}) \\
\end{align*}
\]

Figure 8. Evaluation contexts for Ptolemy, based on [3].

The rules all make implicit use of a fixed (global) list, \( CT \), of the program’s declarations. This list is often implicitly used by auxiliary functions. Several of the rules manipulate type information; this information is not used by the semantics, but is kept for the type soundness proof.

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 \( S'(loc') \neq S(loc') \). The \( \text{fieldsOf} \) 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, \( \text{envOf}(e) \) returns the environment from the current frame \( \nu \), ignoring any other information in \( e \).

\[
\begin{align*}
\text{envOf}(\text{lexframe} \rho \Pi) &= \rho \\
\text{envOf}(\text{evframe} \rho \Pi) &= \rho
\end{align*}
\]

Thus the (VAR) rule says that the value of a variable, including \textbf{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 \( e \) of the receiver object (\( \text{loc} \)), looking in superclasses if necessary, using the auxiliary function \( \text{methodBody} \) (not shown here). The body is executed in a \textbf{alexframe} with an environment that binds the methods formal, including \textbf{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 \( \text{under} \ e \) 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. 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 \textbf{null} to be cast to any type.
Evaluation relation: \[ \Gamma \Rightarrow \Gamma' \]

(NEW) \[ S' = S \oplus (\text{loc} \rightarrow [c. \{ f : \text{null} \mid f \in \text{dom}(\text{fieldsOf}(c)) \}]) \]

(SET) \[ \begin{align*} \rho &= \text{envOf}(v) \quad \rho' &= \rho \circ \var\rightarrow v \quad \Pi = \text{tenvOf}(v) \quad \Pi' = \Pi \cup \var\rightarrow v \circ \text{lexframe}(\rho', \Pi') \end{align*} \]

(CALL) \[ \begin{align*} \{c. F\} &= S(\text{loc}) \quad [\text{evar}\var\rightarrow c] = \text{methodBody}(c, m) \quad \rho' = \{\var\rightarrow v_i \mid 1 \leq i \leq n\} \oplus \{\text{this} \rightarrow v_\var\}
loc &= \text{dom}(S) \quad v' = \text{evframe}(\rho', \Pi') \quad v'' = \text{evframe}(\rho') \quad \theta = \text{pcd}(\var) \quad S' = S \oplus (\text{loc} \rightarrow \text{closure}(H, \theta)(c. \rho, \Pi)) \end{align*} \]

(INVOKE) \[ \begin{align*} \text{closure}((\text{loc}', m, \rho') + H, \theta)(c. \rho', \Pi) &= S(\text{loc}) \\
(c_2, t \leftarrow m(t_1 \var\rightarrow n_1, \ldots, t_n \var\rightarrow n_n)) &= \text{methodBody}(c, m) \\
n &\geq 1 \\
\rho'' &= \rho' \cup \{\var\rightarrow v_\var \mid 1 \leq i \leq n\} \in \{\text{this} \rightarrow v_\var\}
\Pi' &= \Pi \cup \{\var\rightarrow v_\var \mid 1 \leq i \leq n\} \in \{\text{this} \rightarrow v_\var\}
\rho' &= \rho' \cup \{\var\rightarrow v_\var \mid 1 \leq i \leq n\} \in \{\text{this} \rightarrow v_\var\}
\rho'' &= \rho' \cup \{\var\rightarrow v_\var \mid 1 \leq i \leq n\} \in \{\text{this} \rightarrow v_\var\}
\end{align*} \]

Figure 9. Operational semantics of Ptolemy, based on [3].

The (DEF) rule allows for local definitions. It is similar to \texttt{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.

The (NEW) rule simply adds the object being activated from the front of the list of active objects. The bindings in this object are thus given control before others already in the list. Notice that an object can appear in this list multiple times.

\( \text{hbind}(\text{match}(H, J, S, loc)) \)

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

The (EVENT) 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 bindings applicable to the new stack, store, and list of active objects. The resulting list of handler records (\( H \)) is put into an event closure (\text{closure}(H, \theta)(c. \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, \text{invoke} expression that starts running this closure is placed inside an \text{under} expression, so that the stack will be popped when the \text{invoke} expression is finished.

The auxiliary function \( \text{hbind} \), defined in Figure 10 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 (EVENT) rule, the stack passed to it has a new frame on top that represents the current event.

The \text{hmatch} function determines, for a particular object \( \text{loc} \), what bindings declared in the class of the object referred to by \( \text{loc} \) are applicable. It looks up the location \( \text{loc} \) in the store, extracts the class of the object \( \text{loc} \) refers to, and uses that class to obtain a list of potential bindings. This list is filtered using \text{match}, which relies on \text{mpcd} to match a PCD 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 \textit{mpcd}, ultimately from the environment in frames of type \textit{evframe}. This environment is also restricted to contain just those mappings that are for names in the declared formals of the binding.

When a PCD matches the given stack and store, \textit{mpcd} returns an environment, otherwise it returns \text{null}. For named events that match, it returns the environment from the top frame on the stack. For a \textit{cflow} PCD, it searches the stack and returns the first environment that matches the enclosed PCD. The disjunction and conjunction PCDs produce an environment that favors their right argument’s mappings. For disjunction the result is a kind of intersection, and for conjunction the result is a kind of union.

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

The environment is made active by using a \textit{lexframe} containing it as the top frame on the stack. The expression is put inside an \textit{under} expression, so that this new frame will be popped when its evaluation is over.

The \textit{(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 \textit{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 used in a \textit{event} expression.

The operational semantics rules that result in exceptions are given in Figure 11. These treat some uses of null values and bad casts as exceptions, following Java. Encountering one of these exceptions does not make the semantics be “stuck” and hence the situations that lead to these exceptions are not considered to be type errors. However, all of the resulting configurations are terminal.

### 3.4 Type Checking

Type checking uses the type attributes defined in Figure 12. (These use some of the notation and ideas from Schmidt’s book [28].)

The type checking rule themselves are shown in Figure 13 and 14. See Clifton’s thesis [3] for details on these straightforward rules for standard OO expressions. Some rules we use the overriding union notation \(\mathcal{U}\), defined in Figure 10 [28].

As in Clifton’s work [3, 5], 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. For ease of presentation, we also follow Clifton in assuming that the names declared at the top level of a program are distinct and that the extends relation on classes is acyclic.

The type checking of PCDs involves their return type and the typing context (a map from variable names to types) that they make available [3]. The return type and typing context of a named PCD are declared where the event type named is declared. For example, the \texttt{FElement} has \texttt{FElement} as its return type and the typing context that associates \texttt{changedFE} to the type \texttt{FElement}.

Since control flow PCDs are dynamic, their return type cannot be used, so we assign them a return type of \(\top\), which is considered a supertype of \texttt{Object}, but is not legal as a return type itself.

The typing context of a cflow PCD is the typing context of the underlying PCD. Thus if a cflow PCD is not conjoined with any other PCD, the PCD will lead to a type error.

For a disjunction PCD, the return type is the least upper bound of the two PCD’s return types, and the typing context is the intersection of the two typing contexts. For each common names \(I\) that is in the domain of both contexts, the type exposed for \(I\) is the least upper bound of the two types assigned to \(I\) by the two PCDs. This makes sense because only one of the two event types may apply.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11.png}
\caption{Operational semantics of expressions that produce exceptions, based on [3].}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure12.png}
\caption{Type attributes.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure13.png}
\caption{Type-checking rules for OO features.}
\end{figure}
For the conjunction PCD, the return type is the greatest lower bound of the two PCD’s return types, and the typing context is a right-biased overriding union of the two typing contexts. In such an union, each common name Π mapped to the type assigned to Π

By the PCD on right hand side of the conjunction. Note that since a particular PCD must be ultimately based on named event types, and since Ptolemy does not have subtype relationships among named event types, it is usually only sensible to use conjunctions in which one side is not a cfow PCD. When this is done, the return type will be that of the named PCD, since the return type of the cfow PCD is ⊤.

In an event expression, the result type of the body expression, e′, must be a subtype of the result type c declared by the event type, p. Furthermore, the lexical scope available (at e) must provide the context demanded by p.

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 invoke(e).

In the type checking rules above we use several auxiliary functions. Most of these are taken from Clifton’s dissertation [3, Figure 3.3]. A few others are given in Figure 15. The notation τ′ ≼ τ means τ′ is a subtype of τ. It is the reflexive-transitive closure of the declared subclass relationships with the added facts that ⊥ is a supertype of all class type expressions, and that ⊥ is a subtype of all class type expressions. The type ⊥ is used as the type of exceptions. This is formalized in Figure 16.

### Figure 15. Auxiliary functions not in Clifton’s dissertation.

(by the PCD on right hand side of the conjunction. Note that since a particular PCD must be ultimately based on named event types, and since Ptolemy does not have subtype relationships among named event types, it is usually only sensible to use conjunctions in which one side is not a cfow PCD. When this is done, the return type will be that of the named PCD, since the return type of the cfow PCD is ⊤.

In an event expression, the result type of the body expression, e′, must be a subtype of the result type c declared by the event type, p. Furthermore, the lexical scope available (at e) must provide the context demanded by p.

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 invoke(e).

In the type checking rules above we use several auxiliary functions. Most of these are taken from Clifton’s dissertation [3, Figure 3.3]. A few others are given in Figure 15. The notation τ′ ≼ τ means τ′ is a subtype of τ. It is the reflexive-transitive closure of the declared subclass relationships with the added facts that ⊥ is a supertype of all class type expressions, and that ⊥ is a subtype of all class type expressions. The type ⊥ is used as the type of exceptions. This is formalized in Figure 16.

### Figure 16. Subtyping rules, adapted from [3, Figure 3.4].

#### 3.5 Type Soundness

The proof of soundness of Ptolemy’s type system uses a standard preservation and progress argument [37]. The details are adapted from Clifton’s work [3, 5], which in turn follows Flatt et al.’s work [10]. 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 well-typedness:

- For a type d, if d is a supertype of the type of some expression, then the expression is well-typed.
- For a type c, if c is a subtype of the type of some expression, then the expression is well-typed.
- For a type p, if p is a supertype of the type of some expression, then the expression is well-typed.
- For a type r, if r is a subtype of the type of some expression, then the expression is well-typed.
- For a type s, if s is a supertype of the type of some expression, then the expression is well-typed.
- For a type t, if t is a subtype of the type of some expression, then the expression is well-typed.
- For a type u, if u is a supertype of the type of some expression, then the expression is well-typed.
- For a type v, if v is a subtype of the type of some expression, then the expression is well-typed.
- For a type w, if w is a supertype of the type of some expression, then the expression is well-typed.
- For a type x, if x is a subtype of the type of some expression, then the expression is well-typed.
- For a type y, if y is a supertype of the type of some expression, then the expression is well-typed.
- For a type z, if z is a subtype of the type of some expression, then the expression is well-typed.

### Definition 3.1 (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) c(loc) = [e,F], (ii) c ≼ t, (iii) dom(F) = dom(fieldsOf(c)), (iv) rng(F) ⊆ (dom(S) ∪ {null}), and (v) for all f ∈ dom(F), if F(f) = loc′, fields(c)(f) = u, and S(loc′) = [e′,F′], then e′ ≼ u

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

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

**Definition 3.2** (h has type pcd c, π in S). Let h be the handler record \( \langle \text{loc}, \text{m}, \rho \rangle \), let c be a class name, π a type environment, and S a store. Then h has type pcd c, π in S if and only if for some \( c' \), \( c_2, t', n > 1, \text{var}_i, t_i \) and \( \varepsilon: S(\text{loc}) = \{ c, F \} \), methodBody\( (c', m) = (c_2, c \cdot m(\text{var}_1, \ldots, \text{var}_n)) \\{ \varepsilon' \} \), \( \text{dom}(\rho) = \text{dom}(\pi) = \{ \text{var}_2, \ldots, \text{var}_n \} \), \( t_1 = \text{thunk} c \), and for each \( i \in \{ 2, \ldots, n \} \), \( \text{var}_i : \text{var} t_i \in \pi \) and \( \rho(\text{var}) \) 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 \( \nu \), and envOf(\nu) returns \( \nu \)'s environment. Notice that the type environment (EXIT) 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 3.3** (Environment-Stack-Store Consistent). Let \( \Pi \) be a type environment, \( J \) a stack, and \( S \) a store. Then \( \Pi \) is consistent with \( (J, S) \), written \( \Pi \approx (J, S) \), if and only if either \( J = \bullet \) or \( J = \nu + J' \) and for all the following hold:

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

The subject-reduction theorem, as usual, says that evaluation preserves consistency, we must choose \( \Pi \) such that the type environment of frame \( \nu \). Since by hypothesis \( \Pi \vdash e : \text{exp} t \), by the (EVENT EXP TYPE) rule, we have that \( c = t \), and so we can choose \( t' = c \), and thus \( t' \ll t \). With these choices \( \Pi \vdash e : \text{exp} t' \), using the type rules (UNDER EXP TYPE) and (INVOKE EXP TYPE), since \( \Pi \vdash \text{loc} : \text{exp} \text{thunk} c \) by construction. To prove \( \Pi \approx \Pi' \approx (\nu' + \nu + J', S') \) we use definition 3.3. The first condition holds by construction. The second condition holds because the variables in the domain of \( \Pi' \) are a subset of those in the domain of \( \Pi \), and \( \rho' \) is constructed with matching domains, and the only change to \( S' \) from \( S \) is the addition of \( \text{loc} \). The third condition holds because the only location in the domain of \( \Pi' \) is \( \text{loc} \), which has type \( \text{thunk} c \) in \( S' \) by construction.

For the (INVOKE-DONE) rule, suppose \( e = \text{invoke}(\text{loc}) \). From the hypothesis of this rule eClosure(\( \Theta \)) (\( e', \rho', \Pi' \)) = \( S(\text{loc}) \) and \( \nu = \text{lexframe} \rho' \Pi' \nu' \). In this case we have \( e' \ll \text{under} e'' \). From by hypothesis, \( \Pi \vdash \text{invoke}(\text{loc}) : \text{exp} t \), from the (INVOKE EXP TYPE) rule, we have that \( \Pi \vdash \text{loc} : \text{exp} \text{thunk} t \). By hypothesis, we know that \( \Pi \approx (J, S) \), and hence by definition \( \text{loc} \) has type \( \text{thunk} t \) in \( S \). Thus \( \Pi \vdash e' : \text{exp} \text{c''} \), where \( e' \ll t \), so we choose \( t' = e'' \), which makes \( t' \ll t \). It follows directly from the (UNDER EXP TYPE) that \( \Pi'' \vdash e' \ll \text{under} e' : \text{exp} c' \). To prove \( \Pi'' \approx \Pi'' \approx (\nu + J, S') \) we again use definition 3.3. The first condition holds by construction. The second and third conditions hold because of the hypothesis that \( \Pi \approx (J, S) \), since \( \text{loc} \) has type \( \text{thunk} t \) in \( S' \), and thus these conditions hold by parts (b)(iv) and (b)(v) in definition 3.1. For the (INVOKE) rule, suppose \( e = \text{invoke}(\text{loc}) \). From the hypothesis of this rule: eClosure(\( \Theta(\text{loc} \cdot m, \rho + H, \theta \)) (\( e'', \rho'', \Pi'' \)) = \( S(\text{loc}) , [c, F] = \text{loc} \cdot c, c_2, t', n \cdot \text{var}_1, \ldots, \text{var}_n) (e')) = \text{methodBody}(c, m), \rho = \{ \text{var}_1 \mapsto \nu_i, 1 \leq i \leq n \} (\text{this \( \text{loc} \), \Pi'' = \{ \text{var}_i : \text{var} t_i, 1 \leq i \leq n \} (\text{this \( \text{loc} \), \nu', \rho'' \}, \text{loc} \not\in \text{dom}(S), \Pi' = \Pi \vdash \text{loc} : \text{exp} \text{thunk} \text{c} \cdot \text{loc} \cdot \text{eclosure}(H, \theta) (e'', \rho'' \Pi'') \}, \rho_3 \not\in \rho_3 \vdash \{ \text{var}_1 \mapsto \text{loc} \} \)
In Section 2.1, we considered the fragility problem with AO languages. If instead Ptolemy’s event expressions are used to identify events, the ease of selecting join points by showing visual cues at the shadow of event expressions in the module. For example, while Point.setParent(..) is a syntactic pointcut that uses regular expression such as \( \text{set} \{ \ldots \} \), whereas the second pointcut uses the event type \( \text{Point.makeEqual} \) to aggregate all event expressions in the modules that are affected by a change, the developer may choose to restrict or extend the \( \text{Point} \) class as shown in following listing, refactoring of the \( \text{Point} \) class will not have an effect on the events selected by the binding in \( \text{Update} \) class.

```java
class Line extends FigureElement {
    public Point p1, p2;
    public FigureElement setP1(Number x, Number y) {
        Line changedFE = this;
        event Point.makeEqual ( p1.x = x; p1.y = y; )
    }
}
```

Figure 17. The class \( \text{Line} \) in Ptolemy.

For further analysis of robustness against base code changes, let us compare the syntactic version of the pointcuts in the drawing editor example as shown in Figure 2 with Ptolemy’s version in Figure 4.

To remind the reader, the purpose is to expose the abstract state transitions in the \( \text{FigureElement} \) so that aspects can add behaviors at these state transitions. The first pointcut, taken from [36, pp. 56], is a syntactic pointcut that uses regular expression such as \( \text{set} \{ \ldots \} \), whereas the second pointcut uses the event type \( \text{FElement} \) to aggregate all event expressions in the modules that are crosscut by this type-hierarchy.

```java
pointcut Change(FigureElement fe) :
    target(fe) && (call(FElement.set*(...))
    */ Equivalent binding in Ptolemy */
    when FElement do update;

public pointcut Change(FigureElement fe): target(fe) && (call(FElement.set*(...));
```

The syntactic approach to selecting join points provides ease of use. E.g., by just writing a simple regular expression one can select join points throughout the code base. However, this selection is limited to the join points made available by language’s join point model. A much finer-grained selection is possible using our approach; however, systematic modifications are needed to declaratively identify event expressions.

The ease of selecting join points provided by syntactic approaches may turn out to be a double-edged sword. For example, consider another evolutionary scenario. Each composite \( \text{FElement} \) has to be extended to include a reference to the parent \( \text{FElement} \) for ease of traversing the composite structure, e.g. Point is to be extended to include a reference to Line. A mutator \( \text{setParent} \) and an accessor \( \text{getParent} \) for this reference are also added. The syntactic pointcut will also select the join points \( \text{call to mutator setParent for advising, which is incorrect. Setting the reference to the parent, just for ease of implementation, is not an abstract state transition for a FElement. An aspect-oriented tool such as ADTJ may warn the developer against such inadvertent selection of join point by showing visual cues at the shadow of the join point.}

In AspectJ one would exclude calls to setParent by conjuring \( \text{call(FElement+ setParent (...)) to the PCD; however, this solution is not desirable due to two reasons. First, this enumerated list of exceptions can get large in real systems. Second, each item in this list of exception introduces a dependency between the base code and the aspect code, increasing the coupling.}

In Ptolemy, this change will not affect the selected events. The calls to method \( \text{setParent} \) are not automatically selected by the pointcut. However, in cases where the events exposed by a module are affected by a change, the developer may choose to restrict or extend the event expressions in the module. For example, while changing a \( \text{FigureElement} \) subclass to include the methods \( \text{setParent} \) and \( \text{getParent} \), the developer may choose to identify the calls to \( \text{setParent} \) as event expressions.
In summary, it is easier to separate a crosscutting concern using syntactic quantification; however, changes that affect the advised code have a direct impact on the advising code implementation. Some of these impacts may potentially break the advising code. On the other hand, quantified event types require preparation of the code to be advised to systematically provide event expressions. However, advising code is shielded from the changes in advised code by the type-hierarchy. Our approach is thus more robust compared to syntactic quantification against base code changes.

4.3 Flexible Quantification

The event expression in Ptolemy allows one to label any expression as an event expression. Significant flexibility comes from the ability to mark arbitrary expressions, which largely solves the quantification failure problem [33] pointed out in Section 2.2. The events that can be made available to handlers are no longer limited to interface elements. Moreover, the implementations of these events are not exposed to handlers. Handlers only come to rely upon the event type declaration.

4.4 Flexible Access to Context Information

Third problem that we considered in Section 2.3 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 need to provide a mapping from actual context in the lexical scope surrounding the event expression to the context variables made available by the event types. For example, in Figure 4 in the setX method a block expression assigns this to 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 concrete mapping to variables available in the lexical scope of the event expression.

4.5 Uniform Access to Irregular Context Information

Finally, we discussed the inability of the current join point models to provide uniform access to irregular contextual information. An alternative implementation of the example in Figure 3 was presented in Figure 4, where the event expression in setX method and in makeEqual method are given different bindings for the context variable changedFE, however, the handler update was able to access this context information uniformly using the event type name changedFE.

5. Comparative Analysis

In this section, we compare our approach with other similar mechanisms. The mechanisms that we selected for this analysis includes Aspect-Aware Interfaces (AAIs) [19], Open Modules (OMs) [1], and Crosscut Programming Interfaces (XPIs) [33] [36]. Next section summarizes these ideas.

5.1 Overview of Related Ideas

AAIs [19] 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” [19]). 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 program events might be implicitly announced, 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 2 and Section 4.

Aldrich’s proposal on Open Modules [1] is closely related to this work. Both approaches have two similar advantages. First, like our work, open modules also allows a class developer to explicitly expose pointcuts for behavioral modifications by aspects. The implementations of these pointcuts remain hidden from the aspects. As a result, the impact of base code changes on the aspect is reduced. Second, with appropriate language extensions, an explicitly exposed pointcut may also expose the right contextual information uniformly across the join points selected by the pointcut. However, OMs exacerbates the problem of quantification failure. Each explicitly declared pointcut has to be enumerated by the aspect for advising. On the other hand, our approach significantly simplifies quantification. Instead of manually enumerating the join points of interest, one can use the crosscutting type-hierarchy for implicit non-syntactic selection of join points.

Similar to OMs, a programmer using Ptolemy’s event types must systematically modify modules in a system that a given concern crosscuts to expose join points that are to be advised, by using event expressions. For example, the modules Line, Point, etc. were modified to expose join points of type FEChange. However, unlike OMs, once these modules have incorporated such event expressions, no awkward enumeration of explicitly exposed join points is necessary for quantification. Instead, one simply uses the event type FEChange in a PCD.

Sullivan et al. [33] 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 [36]). These design rule interfaces were later called crosscut programming interface (XPI) by Griswold et al. [36]. 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.

5.2 Metrics and Analysis Results

The criteria and the analysis results are summarized in Figure 18. The rest of this section presents our analysis in detail.

5.2.1 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 possible 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 provides a two-way abstraction for all context information exchanged between an event expression and the handler.
<table>
<thead>
<tr>
<th>Metrics</th>
<th>Description</th>
<th>AAIs</th>
<th>OMs</th>
<th>XPIs</th>
<th>Polytool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>Does the mechanisms support abstraction?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Aspect/Base IH</td>
<td>Is information hiding supported for aspect / base?</td>
<td>Aspect</td>
<td>Base</td>
<td>Aspect + Base</td>
<td>Aspect + Base</td>
</tr>
<tr>
<td>Reasoning</td>
<td>What is the granularity of reasoning?</td>
<td>Join point</td>
<td>Module</td>
<td>XPIs Scope</td>
<td>Join point</td>
</tr>
<tr>
<td>Configuration</td>
<td>Does it require complete system configuration?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Decoupling</td>
<td>Does it decouple aspect from base code?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Locality</td>
<td>Are the interface definitions textually localized?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Stable</td>
<td>Is it stable against code changes?</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Pattern</td>
<td>Does it allow pattern-based quantification?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Does it allow quantification based on type-hierarchy?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Scope</td>
<td>Is the scope of the interface?</td>
<td>Program</td>
<td>Module</td>
<td>User defined</td>
<td>User defined</td>
</tr>
<tr>
<td>Scope control</td>
<td>Is fine-grained control over scope available?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Adaptation</td>
<td>Does it require base code adaption/?refactoring?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Observable</td>
<td>Is it pure oblivious?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Lexical hints</td>
<td>Does it provides lexical hints in a module?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 18. Results of comparative analysis

5.2.2 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.

5.2.3 Locality

This criterion evaluates whether the AO interface definitions are textually localized. AAIs are computed once per place in the code where advice might apply, and thus are not localized. OMs are also 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 the case of Ptolemy, the event expressions are not localized but the type definition that serves as an interface to the handlers is localized.

5.2.4 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 `withinline`. 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 `withinline` 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 language’s expressiveness.

5.2.5 Base Code Adaptation and Obliviousness

Obliviousness is a widely accepted tenet for aspect-oriented software development [9]. 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 [1, 4, 7, 36, 19, 30, 33]. To understand the behavior of a module in the presence of aspects and for independent evolution of base and aspect code, it is necessary to understand all applicable aspects. Tools such as AspectJ Development Tools alleviate the problem, but not completely.

According to Sullivan et al.[33], there are many variants of the notion of obliviousness, language-level obliviousness, feature obliviousness, designer obliviousness, and pure obliviousness. Language-level obliviousness comes from introducing quantification mechanisms in the language. Feature obliviousness is when the designer of the base code is aware of the presence of aspects but unaware of the features that the aspect implements. Designer obliviousness comes when the base code designer can be unaware of the presence of an aspect. Pure obliviousness is when both base and aspect code designers are symmetrically unaware of each other.

None of these approaches support pure obliviousness. AAIs support language-level obliviousness, feature obliviousness, and designer obliviousness. Designers of base code are aware of the presence of aspects advising piece of base code but need not be aware of the exact feature that these aspects implement nor do they need to prepare base code. OMs support language-level obliviousness and feature obliviousness but not the designer obliviousness. Ptolemy’s design discards designer obliviousness. The base code designers have to prepare their code by exposing desired events. However, similar to XPIs [33, 36] it preserves feature obliviousness. The base code designers can be completely unaware of “spectators” [4] or “harmless” aspects [7] that advise them.

In our drawing example, Point and Line expose events of type FEChange without being aware of the actual usage of the event. In the example, the event was used for modularizing the display update policy, but it could have been used for modularizing other concerns as well. For example, a persistence policy that requires updating a persistent representation of Point and Line can also be implemented. All such observers may be implemented simultaneously as different modules without Point and Line designers being aware of them and without these observers being dependent on the details of Point and Line.

6. Other Related Ideas

Explicitly identifying join points is not a new idea, e.g. it has appeared previously in SetPoint [2], the notion of type-based quantification [24] and the notion of implicit invocation with implicit announcement [31]; however, the novelty of our approach lies in providing explicitly declared join points with types with a sound type system. As in SetPoint, explicitly declared event expressions provide more robust quantification with respect to base code changes, and declared event types provide precise interfaces between han-
Similar ideas have also been independently investigated by Steinmann and Pawlitzki [31]. Their language also has event types. These are similar to Ptolemy’s event types, but are used differently. In that event announcement leads to the creation of an object of the event type, with the declared context realized as fields of that object. Their language is a modification of AspectJ, and has both implicit (PCD-based) and explicit announcement of events, whereas Ptolemy only has explicit announcement. In their language explicit announcement passes context positionally, whereas in Ptolemy context is not positional. 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. They have a prototype implementation, but do not formally present their language’s type system.

Delegates in .NET languages such as C# and EventObject class in Java standard library 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 event types.

Some approaches provide new pointcut expressions to select statement and expression level join points for advising [14, 27]. Compared to these proposals, our approach provides two benefits. First, the body of an event expression is hidden from the design of the handlers by a typed interface. The handler and PCD are not coupled with the encapsulated details of the events, only with the event type. Second, an event expression provides textual hints to the developer, in the module code itself that may reduce unintentional impacts of the base code changes on the handlers.

Another related area is mediator-based design styles [34]. In this design style, in addition to providing methods that can be called, modules declare and announce events. Other modules can register operations to be invoked by events. An invocation relation would be perfectly sensible to allow event types to declare that object. Their language is a modification of AspectJ, and has some advantages, for example, base code does not need to know the names used in event type declarations. However, computing parameter lists for conjunctions and disjunctions of PCDs in a positional design seems less intuitive than in Ptolemy.

Another change from AspectJ is that Ptolemy’s invoke does not allow one to change the context available to the next handler or to the original event expression. In AspectJ, one can change this context when using invoke. We believe that this feature is orthogonal to the main points of our work on Ptolemy, and hence omitted this feature of the semantics. Nevertheless, not allowing writable context should make it easier to reason about code in Ptolemy, since Ptolemy’s handlers thus have fewer control effects.

7.2 Possible Extensions

Before and after advice, as in AspectJ, is easily simulated with around advice. However, their typing is slightly different, as before and after advice do not have a return type [3, 5, 17].

AspectJ also allows negation in its syntax for PCDs. We could type check such PCDs by using $T$ as the return type (which we also do for cflow PCDs), and by using an empty typing context. Thus negation PCDs could only be used in conjunction with other PCDs.

Event closures in Ptolemy are a second-class feature, since they cannot be stored in object fields, but can only be passed around as arguments and temporarily stored in local variables. This makes it possible to stack allocate event closures, and also makes for fewer control effects. However, it would be interesting to see what kind of additional power one gets with first-class event closures.

Ptolemy does not have subtyping of event types. However, it would be perfectly sensible to allow event types to declare that they extend other event types that have the same return type, with inheritance of the context declarations of their supertypes. This would allow an event expression having more than one type, and would make it sensible to have a lattice of event types.

7.3 Non-Nested Overlap of Event Expressions?

Consider a sequence expression such as $e_1; e_2; e_3$. One might think that it would be possible that one part of a program might need to advise $e_1; e_2$ and that another part of the program might need to advise $e_2; e_3$. However, this is not syntactically possible in Ptolemy, since an event expression must encompass an entire expression as its body, and a sequence such as $e_1; e_2; e_3$ must be either $e_1; (e_2; e_3)$ or $(e_1; e_2); e_3$, but not both.
8. Conclusion and Future Work

The main contribution of this work is the notion of quantified, typed events. Our event types contain information about the names and types of exposed context. In our new event model, events can be identified declaratively by attaching an event type name to an expression. In particular, we showed that quantified, typed events improve the robustness of the handler code against base code changes, and makes it easier for handlers to uniformly access reflective information about the events without breaking encapsulation.

Our proposal offers new directions to investigate. One would be to combine our type system with an effect system. Effect declarations could be used to limit the potential side effects of handler methods [3, 6], which would allow more efficient reasoning about them. 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.

Acknowledgments

Rajan was supported in part by the NSF grant CNS-0627354. Leavens was supported in part by NSF grant CCF-0429567. Both were supported in part by NSF grant CNS-07-09217. The discussions with participants of the seminar course Com S 610-HR, in particular, with Juri Memmert were very helpful. Thanks to Friedrich Steimann for discussions about these ideas. Thanks to Paulo Borba and several POPL 2008 PC members for helpful comments on earlier versions of this paper.

References

[16] A. Igarashi, B. Pierce, and P. Waldner. Featherweight Java: A minimal core calculus for Java and GJ. In OOPSLA ’99, pages 132–146.