

Ptolemy: A Language of Quantified, Typed Events

Hridesh Rajan and Gary T. Leavens

TR #07-13

Initial Version: July 26, 2007. Revised: October 4, 2007.

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 © 2007, Hridesh Rajan and Gary T. Leavens.

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

Ptolemy: A Language of Quantified, Typed Events

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

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

```
1 class FElement extends Object{}
2 class Point extends FElement { /* ... */
3   Number x; Number y;
4   FElement setX(Number x) {
5     Point changedFE = this;
6     event FEChange {
7       this.x = x; this
8     }
9   }
10  FElement makeEqual(Point other) {
11    Point changedFE = other;
12    event FEChange {
13      other.x = this.x; other.y = this.y;
14      other
15    }
16  }
17 }
18 FElement evtype FEChange {FElement changedFE;}
19 class Update extends Object {
20   FElement last;
21   Update init() { register(this) }
22   FElement update(thunk FElement next,
23                   FElement changedFE) {
24     this.last = changedFE;
25     Display.update();
26     proceed(next)
27   }
28   FElement around(FElement changedFE)
29     FEChange:update
30 }
```

Figure 1. Drawing Editor in Ptolemy

Abstract

This paper defines Ptolemy. The novelty of Ptolemy is the notion of event types and quantification based on event types. We give the syntax, operational semantics and type rules for the language, and discuss its meta-theory.

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

1. Ptolemy’s Design

In this section, we describe Ptolemy, a language with quantified, typed events 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 1.

1.1 Overview

Ptolemy is inspired by II languages such as Rapide [8] and AO languages such as AspectJ [5]. It also incorporates some ideas from Eos [11] and Caesar [9]. As a small, core language, its technical presentation shares much in common with Clifton and Leavens’s MiniMAO₁ [1, 2], which itself builds on Classic Java [4] and Featherweight Java [7]. The object-oriented part of Ptolemy follows MiniMAO₀. While it has classes, objects, inheritance, and subtyping, 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: an event type declaration (**evtype**), and an event expression (**event**).

Like Eos [11], 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 a proceed closure as its first argument. A *proceed closure* [11] (or delegate, or delegate chain) contains code needed to run the applicable handlers and the original event expression. A proceed closure can be run using a **proceed** expression.

Like II languages a Ptolemy module can register to receive event notifications. This capability is the same as “deployment” in the AO languages Eos and Caesar [9]. 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 [12]. 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.

1.2 Syntax

Ptolemy’s syntax is shown in Figure 2 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.

1.2.1 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

```

prog ::= decl* e
decl ::= class c extends d { field* meth* binding* }
      | c evtype p { form* }
field ::= c f;
meth ::= t m (form*) { e }
t ::= c | thunk c
binding ::= c around (form*) pcd : m
form ::= t var, where var ≠ this
pcd ::= p | cflow (pcd) | pcd && pcd | pcd ' | ' pcd
e ::= new c () | var | null | e.m (e*) | e.f | e.f = e
    | cast c e | form = e; e | e; e
    | register (e) | event p { e } | proceed (e)

c, d ∈ C, the set of class names
p ∈ P, the set of evtype names
f ∈ F, the set of field names
m ∈ M, the set of method names
var ∈ {this} ∪ V, V is the set of variable names

```

Figure 2. Ptolemy’s Abstract syntax, based on [1, Figure 3.1, 3.7].

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 1, which contains a binding on lines 28–29. 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 1 on line 18. In writing examples of event types, as in Figure 1, 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 result type `FEElement`, 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 `FEElement` 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 1 (lines 5–8).

1.2.2 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 1, 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.

1.2.3 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** $c()$), variable dereference (var , including **this**), field dereference ($e.f$), **null**, cast (**cast** $t e$), assignment to a field ($e_1.f = e_2$), a definition block ($t var = e_1; e_2$), and sequencing ($e_1; e_2$). Their semantics and typing is fairly standard [1, 2].

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

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 1 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 [6] of an event of type p . Note that only (well-formed) expressions can produce events, one may not, describe 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 5, and (ii) is used in type checking.

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

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

An example demonstrating these features is shown in Figure 1. 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 1 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 event expression 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 a pro-

ceed closure, which also remembers the event expression’s body. Then the first handler, if any, is run; if it proceeds, it will run the next handler, or the body expression if there are no more handlers.

This ordering of handlers in the proceed closure is designed to allow more recently registered objects to control whether previously registered objects have their handlers run, by using **proceed** (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 1 followed by the main expression

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

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`, a proceed 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 proceed closure (the expression `proceed(next)`), which executes the body of the event expression (starting at line 7 of Figure 1) 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 expression is as follows.

$$\begin{aligned} & \text{event } p_1, \dots, p_n \{e\} \\ \Rightarrow & \text{event } p_1 \{ \dots \text{event } p_n \{e\} \dots \} \end{aligned}$$

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

1.3 Operational Semantics

This section defines a small step operational semantics for Ptolemy. This semantics has been implemented in the logic programming language λ Prolog, using the Teyjus system [10]. This semantics and its description in this section is adapted from Clifton’s work [1, 2, 3], which builds on Classic Java [4]. 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 3. 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 ::= loc | under e | NullPointerException | ClassCastException
loc ∈ L, the set of locations
```

Figure 3. 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 4. A configuration contains an expression (e), a 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 of the event type being run. A value is a location or **null**. Stores are maps from locations to storable values. Storable values are either objects or proceed closures. Objects have a class and also a map from field names to values. Proceed closures (**proceedClosure**) 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 9) 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 (loc), a method name (m), and an environment (ρ_h). The environment ρ_h is used to assemble the method call arguments when the handler method is called. The environment ρ recorded at the top level of the proceed closure is used to run the expression e when a proceed closure with no handler records is used in a **proceed** expression.

Domains:

$\Gamma ::= \langle e, J, S, A \rangle$	“Configurations”
$J ::= \nu + J \mid \bullet$	“Stacks”
$\nu ::=$	“Frames”
lexframe $\rho \Pi$	“Lexical”
evframe $p \rho \Pi$	“Event execution”
$\rho ::= \{j : v_k\}_{k \in K},$	“Environments”
where K is finite, $K \subseteq I$	
$v ::= loc \mid \text{null}$	“Values”
$S ::= \{loc_k \mapsto sv_k\}_{k \in K},$	“Stores”
where K is finite	
$sv ::= o \mid pc$	“Storable Values”
$o ::= [c, F]$	“Object Records”
$F ::= \{f_k \mapsto v_k\}_{k \in K},$	“Field Maps”
where K is finite	
$pc ::= \text{proceedClosure}(H, \theta)(e, \rho, \Pi)$	“Proceed 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”

Figure 4. Domains used in the semantics, based on [1].

As is usual [14] the semantics is presented as a set of evaluation contexts \mathbb{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 5 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 $\langle \text{under } e, (\text{lexframe } \{\} \{\}) + \bullet, \{\}, \bullet \rangle$, which starts evaluation of e in a frame with an empty environment, and with an empty store and empty list of active objects.

Evaluation contexts:

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

Figure 5. Evaluation contexts for Ptolemy, based on [1].

Figure 6 presents the operational semantics of Ptolemy. In these rules all of the hypotheses are really side conditions and side definitions for use in the rule. 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 $S'(loc') = S(loc')$. 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 .

$$\begin{aligned} envOf(\mathbf{lexframe} \ \rho \ \Pi) &= \rho \\ envOf(\mathbf{evframe} \ p \ \rho \ \Pi) &= \rho \end{aligned}$$

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. 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.

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. 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 a proceed closure ($\mathbf{proceedClosure}(H, \theta)(e, \rho', \Pi)$), which is placed in the store at a fresh location. This proceed 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,

Evaluation relation: $\hookrightarrow: \Gamma \rightarrow \Gamma$

$$\begin{aligned} & \text{(NEW)} \\ & \frac{loc \notin dom(S)}{S' = S \oplus (loc \mapsto [c.\{f \mapsto \mathbf{null} \mid f \in dom(fieldsOf(c))\}])} \\ & \frac{}{\langle \mathbb{E}[\mathbf{new} \ c()], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[loc], J, S', A \rangle} \\ & \text{(VAR)} \quad \frac{\rho = envOf(\nu) \quad v = \rho(var)}{\langle \mathbb{E}[var], \nu + J, S, A \rangle \hookrightarrow \langle \mathbb{E}[v], \nu + J, S, A \rangle} \quad \text{(GET)} \quad \frac{[c.F] = S(loc) \quad v = F(f)}{\langle \mathbb{E}[loc.f], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[v], J, S, A \rangle} \\ & \text{(SET)} \quad \frac{[c.F] = S(loc) \quad 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} \\ & \text{(DEF)} \quad \frac{\rho = envOf(\nu) \quad \rho' = \rho \oplus (var \mapsto v) \quad \Pi = tenvOf(\nu) \quad \Pi' = \Pi \uplus \{var : \mathbf{var} \ t\} \quad \nu' = \mathbf{lexframe} \ \rho' \ \Pi'}{\langle \mathbb{E}[t \ \mathbf{var} = v; e], \nu + J, S, A \rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under} \ e], \nu' + \nu + J, S, A \rangle} \\ & \text{(CALL)} \quad \frac{[c.F] = S(loc) \quad (c_2, t \ m(t_1 var_1, \dots, t_n var_n)\{e\}) = methodBody(c, m) \quad \rho = \{var_i \mapsto v_i \mid 1 \leq i \leq n\} \oplus \{\mathbf{this} \mapsto loc\} \quad \Pi = \{var_i : \mathbf{var} \ t_i \mid 1 \leq i \leq n\} \uplus \{\mathbf{this} : \mathbf{var} \ c_2\} \quad \nu = \mathbf{lexframe} \ \rho \ \Pi}{\langle \mathbb{E}[loc.m(v_1, \dots, v_n)], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under} \ e], \nu + J, S, A \rangle} \\ & \text{(CAST)} \quad \frac{[c'.F] = S(loc) \quad c' \preceq c}{\langle \mathbb{E}[\mathbf{cast} \ c \ loc], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[loc], J, S, A \rangle} \quad \text{(SKIP)} \quad \frac{}{\langle \mathbb{E}[v; e], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[e], J, S, A \rangle} \\ & \text{(UNDER)} \quad \frac{}{\langle \mathbb{E}[\mathbf{under} \ v], \nu + J, S, A \rangle \hookrightarrow \langle \mathbb{E}[v], J, S, A \rangle} \quad \text{(REGISTER)} \quad \frac{}{\langle \mathbb{E}[\mathbf{register}(loc)], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[loc], J, S, loc + A \rangle} \\ & \text{(EVENT)} \quad \frac{\rho = envOf(\nu) \quad \Pi = tenvOf(\nu) \quad (c \ \mathbf{evtype} \ p\{t_1 var_1, \dots, t_n var_n\}) \in CT \quad \rho' = \{var_i \mapsto v_i \mid \rho(var_i) = v_i\} \quad \pi = \{var_i : \mathbf{var} \ t_i \mid 1 \leq i \leq 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{proceedClosure}(H, \theta)(e, \rho, \Pi))}{\langle \mathbb{E}[\mathbf{event} \ p \ \{e\}], \nu + J, S, A \rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under} \ (\mathbf{proceed}(loc))], \nu' + \nu + J, S', A \rangle} \\ & \text{(PROCEED-DONE)} \quad \frac{\mathbf{proceedClosure}(\bullet, \theta)(e, \rho, \Pi) = S(loc) \quad \nu = \mathbf{lexframe} \ \rho \ \Pi}{\langle \mathbb{E}[\mathbf{proceed}(loc)], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under} \ e], \nu + J, S, A \rangle} \\ & \text{(PROCEED-RUN)} \quad \frac{\mathbf{proceedClosure}((loc', m, \rho) + H, \theta)(e, \rho', \Pi) = S(loc) \quad [c.F] = S(loc') \quad (c_2, t \ m(t_1 var_1, \dots, t_n var_n)\{e'\}) = methodBody(c, m) \quad n \geq 1 \quad \rho'' = \{var_i \mapsto v_i \mid 2 \leq i \leq n, v_i = \rho(var_i)\} \quad loc_1 \notin dom(S) \quad S' = S \oplus (loc_1 \mapsto \mathbf{proceedClosure}(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 \leq i \leq n\} \uplus \{\mathbf{this} : \mathbf{var} \ c_2\} \quad \nu = \mathbf{lexframe} \ \rho''' \ \Pi'}{\langle \mathbb{E}[\mathbf{proceed}(loc)], J, S, A \rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under} \ e'], \nu + J, S', A \rangle} \end{aligned}$$

Figure 6. Operational semantics of Ptolemy, based on [1].

proceed expression that starts running this closure is placed inside an **under** expression, so that the stack will be popped when the proceed expression is finished.

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

$hbind(J, S, \bullet) = \bullet$
 $hbind(J, S, loc + A)$
 $= concat(hmatch(CT, J, S, loc), hbind(J, S, A))$
 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 $bindings(CT, c) = H$

$bindings(CT, c) = binds(CT, CT, c)$
 $binds(CT, \bullet, c) = \bullet$
 $binds(CT, ((t \text{ evtype } p\{\dots\}) + CT'), c) = binds(CT, CT', c)$
 $binds(CT, ((\text{class } c \text{ extends } c' \dots binding_1 \dots binding_n) + CT'), c)$
 $= concat((binding_n + \dots + binding_1 + \bullet), binds(CT, CT', c))$

$match(\bullet, J, S, loc) = \bullet$
 $match(binding + H, J, S, loc)$
 $= \text{if } mpcd(pcd, J, S) \neq \perp$
 $\quad \text{then let } \rho = mpcd(pcd, J, S)$
 $\quad \quad \text{in let } \rho' = \{var_i \mapsto \rho(var_i) \mid 1 \leq i \leq n\}$
 $\quad \quad \text{in } (\text{loc}, m, \rho') + match(H, J, S, loc)$
 $\quad \text{else } match(H, J, S, loc)$
 where $binding = t_1 \text{ around}(t_1 \text{ var}_1, \dots, t_n \text{ var}_n) pcd : m$

$mpcd(p, (\text{evframe } p' \rho \Pi) + J, S) = \text{if } p \equiv p' \text{ then } \rho \text{ else } \perp$
 $mpcd(\text{cflow } pcd, \bullet, S) = \perp$
 $mpcd(\text{cflow } pcd, \nu + J, S)$
 $= \text{if } mpcd(pcd, \nu + J, S) \neq \perp \text{ then } mpcd(pcd, \nu + J, S)$
 $\quad \text{else } mpcd(pcd, J, S)$
 $mpcd(pcd_1 \&\& pcd_2, J, S)$
 $= \text{if } mpcd(pcd_1, J, S) \neq \perp \wedge mpcd(pcd_2, J, S) \neq \perp$
 $\quad \text{then } mpcd(pcd_1, J, S) \cup mpcd(pcd_2, J, S)$
 $\quad \text{else } \perp$
 $mpcd(pcd_1 \parallel pcd_2, J, S)$
 $= \text{if } mpcd(pcd_1, J, S) \neq \perp \wedge mpcd(pcd_2, J, S) \neq \perp$
 $\quad \text{then } mpcd(pcd_1, J, S) \cap mpcd(pcd_2, J, S)$
 $\quad \text{else if } mpcd(pcd_1, J, S) \neq \perp \text{ then } mpcd(pcd_1, J, S)$
 $\quad \text{else } mpcd(pcd_2, J, S)$

$\rho \cup \rho' = \{var \mapsto \rho(var) \mid var \in dom(\rho) \wedge var \notin dom(\rho')\} \cup \rho'$
 $\rho \cap \rho' = \{var \mapsto \rho'(var) \mid var \in dom(\rho) \wedge var \in dom(\rho')\}$

Figure 7. Auxiliary functions for matching bindings.

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 $hmatch$ 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 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 $mpcd$, 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 a PCD matches the given stack and store, $mpcd$ returns an environment, otherwise it returns \perp . For named events that match, it returns the environment from the top frame on the stack. For a **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 **proceed** expressions is done by the two proceed rules. The (PROCEED-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 proceed closure. The environment is made active by

(NCALL)
 $\langle \mathbb{E}[\text{null}.m(v_1, \dots, v_n)], J, S, A \rangle \hookrightarrow \langle \text{NullPointerException}, \bullet, S, A \rangle$

(NGET)
 $\langle \mathbb{E}[\text{null}.f], J, S, A \rangle \hookrightarrow \langle \text{NullPointerException}, \bullet, S, A \rangle$

(NSET) (NCAST)
 $\langle \mathbb{E}[\text{null}.f = v], J, S, A \rangle \hookrightarrow \langle \text{NullPointerException}, \bullet, S, A \rangle$ $\langle \mathbb{E}[\text{cast } t \text{ null}], J, S, A \rangle$
 $\hookrightarrow \langle \text{NullPointerException}, \bullet, S, A \rangle$ $\hookrightarrow \langle \mathbb{E}[\text{null}], J, S, A \rangle$

(XCAST)

$$\frac{[c.F] = S(loc) \quad c \not\approx t}{\langle \mathbb{E}[\text{cast } t \text{ loc}], J, S, A \rangle \hookrightarrow \langle \text{ClassCastException}, \bullet, S, A \rangle}$$

(NREGISTER)
 $\langle \mathbb{E}[\text{register null}], J, S, A \rangle \hookrightarrow \langle \text{NullPointerException}, \bullet, S, A \rangle$

Figure 8. Operational semantics of expressions that produce exceptions, based on [1].

using a **lexframe** containing it as the top frame on the stack. The expression is put inside an **under** expression, so that this new frame will be popped when its evaluation is over.

The (PROCEED-RUN) rule handles the case where there are handler records still to be run in the proceed 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 proceed closure that would run the rest of the handler records (and the original event's body) if it used in a **proceed** expression.

The operational semantics rules that result in exceptions are given in Figure 8. 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.

1.4 Type Checking

Type checking uses the type attributes defined in Figure 9. (These use some of the notation and ideas from Schmidt's book [13].)

$\theta ::=$	“type attributes”
OK	“program/top-level decl.”
OK in c	“method, binding”
var t	“var/formal/field”
exp t	“expression”
pcd τ, π	“pcd/handler chain”
$\tau ::= c \mid \top \mid \perp$	“class type exps”
$\pi, \Pi ::= \{I : \theta_I\}_{I \in K}$,	“type environments”
where K is finite, $K \subseteq (\mathcal{L} \cup \{\text{this}\}) \cup \mathcal{V}$	

Figure 9. Type attributes.

The type checking rule themselves are shown in Figure 10 and 11. See Clifton's thesis [1] for details on these straightforward rules for standard OO expressions. Some rules we use the overriding union notation \cup , defined in Figure 7 [13]. As in Clifton's work [1, 2], 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 [1]. The return type and typing context of a named PCD are declared where the event type named is declared. For

$\frac{\text{(NEW EXP TYPE)}}{\text{isClass}(c)} \quad \frac{\text{(CAST EXP TYPE)}}{\text{isClass}(c)} \\ \Pi \vdash \mathbf{new} c() : \mathbf{exp} c \quad \Pi \vdash \mathbf{cast} c e : \mathbf{exp} c$	
$\frac{\text{(NULL EXP TYPE)}}{\text{isClass}(c)} \quad \frac{\text{(GET EXP TYPE)}}{\Pi \vdash e : \mathbf{exp} c \quad \text{fieldsOf}(c)(f) = t} \\ \Pi \vdash \mathbf{null} : \mathbf{exp} c \quad \Pi \vdash e.f : \mathbf{exp} t$	
$\frac{\text{(SET EXP TYPE)}}{\Pi \vdash e : \mathbf{exp} c \quad \text{fieldsOf}(c)(f) = t \quad \Pi \vdash e' : \mathbf{exp} t' \quad t' \preceq t} \\ \Pi \vdash e.f = e' : \mathbf{exp} t'$	
$\frac{\text{(DEF EXP TYPE)}}{\text{isType}(t) \quad \Pi \vdash e_1 : \mathbf{exp} t_1} \\ t_1 \preceq t \quad \Pi' = \Pi \uplus \{ \mathbf{var} : \mathbf{var} t \} \quad \Pi' \vdash e_2 : \mathbf{exp} t_2 \\ \Pi \vdash t \mathbf{var} = e_1; e_2 : \mathbf{exp} t_2$	
$\frac{\text{(SEQ EXP TYPE)}}{\Pi \vdash e_1 : \mathbf{exp} t_1 \quad \Pi \vdash e_2 : \mathbf{exp} t_2} \\ \Pi \vdash e_1; e_2 : \mathbf{exp} t_2$	
$\text{(NP EXCEPTION EXP TYPE)} \\ \Pi \vdash \mathbf{NullPointerException} : \mathbf{exp} \perp$	
$\text{(CC EXCEPTION EXP TYPE)} \\ \Pi \vdash \mathbf{ClassCastException} : \mathbf{exp} \perp$	

Figure 10. Type-checking rules for OO features.

example, the FEChange PCD has FEElement as its return type and the typing context that associates changedFE to the type FEElement.

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 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.

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 a union, each common name I mapped to the type assigned to I 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 cflow PCD. When this is done, the return type will be that of the named PCD, since the return type of the cflow PCD is \top .

In an event 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 lexical scope available (at e) must provide the context demanded by p .

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

In the type checking rules above we use several auxiliary functions. Most of these are taken from Clifton's dissertation [1, Figure 3.3]. A few others are given in Figure 12.

$\frac{\text{(CHECK PROGRAM)}}{(\forall i \in \{1..n\} :: \vdash \text{decl}_i : \text{OK}) \quad \vdash e : \mathbf{exp} t} \\ \vdash \text{decl}_1 \dots \text{decl}_n e : \mathbf{prog} t$	
$\frac{\text{(CHECK CLASS)}}{\text{isClass}(d) \quad (\forall j \in \{1..m\} :: \vdash \text{meth}_j \text{OK in } c) \\ (\forall k \in \{1..o\} :: \vdash \text{binding}_k \text{OK in } c) \\ (\forall i \in \{1..n\} :: \text{isClass}(t_i) \wedge f_i \notin \text{dom}(\text{fieldsOf}(d)))} \\ \vdash \mathbf{class} c \text{ extends } d \{ t_1 f_1; \dots t_n f_n; \text{meth}_1 \dots \text{meth}_m \\ \text{binding}_1 \dots \text{binding}_o \} : \text{OK}$	
$\frac{\text{(CHECK EVTYPE)}}{\text{isClass}(c) \quad (\forall i \in \{1..n\} :: \text{isType}(t_i))} \\ \vdash c \mathbf{evtype} p \{ t_1 \text{var}_1; \dots t_n \text{var}_n \} : \text{OK}$	
$\frac{\text{(CHECK METHOD)}}{\text{isType}(t) \quad (\forall i \in \{1..n\} :: \text{isType}(t_i))} \\ \{ \text{var}_1 : \mathbf{var} t_1, \dots, \text{var}_n : \mathbf{var} t_n, \mathbf{this} : \mathbf{var} c \} \vdash e : \mathbf{exp} t' \\ t' \preceq t \quad (\mathbf{class} c \text{ extends } d \{ \dots \}) \in CT \\ \text{override}(m, d, t_1 \times \dots \times t_n \rightarrow t)} \\ \vdash t m(t_1 \text{var}_1, \dots, t_n \text{var}_n) \{ e \} : \text{OK in } c$	
$\frac{\text{(CHECK BINDING)}}{\text{isClass}(c') \quad n \geq 1} \\ t_1 = \mathbf{thunk} c' \quad (\forall i \in \{2..n\} :: \text{isType}(t_i)) \quad \vdash \text{pcd} : \mathbf{pcd} c', \pi \\ (c_2, c' m(t_1 \text{var}_1, \dots, t_n \text{var}_n) \{ e \}) = \text{methodBody}(c, m) \\ \{ \text{var}_2 : \mathbf{var} t_2, \dots, \text{var}_n : \mathbf{var} t_n \} \subseteq \pi} \\ \Pi \vdash (c' \mathbf{around}(t_1 \text{var}_1, \dots, t_n \text{var}_n) \text{pcd} : m) : \text{OK in } c$	
$\frac{\text{(EV ID PCD TYPE)}}{(c \mathbf{evtype} p \{ t_1 \text{var}_1; \dots t_n \text{var}_n \}) \in CT \\ \pi = \{ \text{var}_1 : \mathbf{var} t_1, \dots \text{var}_n : \mathbf{var} t_n \}} \\ \vdash p : \mathbf{pcd} c, \pi$	
$\frac{\text{(CFLOW PCD TYPE)}}{\vdash \text{pcd} : \mathbf{pcd} \tau, \pi} \\ \vdash \mathbf{cflow}(pcd) : \mathbf{pcd} \top, \pi$	$\text{(CONJUNCTION PCD TYPE)} \\ \vdash \text{pcd} : \mathbf{pcd} \tau, \pi \\ \vdash \text{pcd}' : \mathbf{pcd} \tau', \pi' \\ \tau'' = \tau \sqcup \tau' \quad \pi'' = \pi \uplus \pi' \\ \vdash \text{pcd} \&\& \text{pcd}' : \mathbf{pcd} \tau'', \pi''$
$\frac{\text{(DISJUNCTION PCD TYPE)}}{\vdash \text{pcd} : \mathbf{pcd} \tau, \pi} \\ \vdash \text{pcd}' : \mathbf{pcd} \tau', \pi' \\ \tau'' = \tau \sqcap \tau' \quad \pi'' = \pi \cap \pi' \\ \vdash \text{pcd} \text{pcd}' : \mathbf{pcd} \tau'', \pi''$	$\text{(VAR EXP TYPE)} \\ (\text{var} : \mathbf{var} t) \in \Pi \\ \Pi \vdash \text{var} : \mathbf{exp} t$
$\frac{\text{(CALL EXP TYPE)}}{\Pi \vdash e : \mathbf{exp} c} \\ (c_2, t m(t_1 \text{var}_1, \dots, t_n \text{var}_n) \{ e \}) = \text{methodBody}(c, m) \\ c \preceq c_2 \quad \Pi \vdash e_1 : \mathbf{exp} t_1 \dots \Pi \vdash e_n : \mathbf{exp} t_n} \\ \Pi \vdash e.m(e_1, \dots, e_n) : \mathbf{exp} t$	
$\frac{\text{(EVENT EXP TYPE)}}{(c \mathbf{evtype} p \{ t_1 \text{var}_1; \dots t_n \text{var}_n \}) \in CT \\ \{ \text{var}_1 : \mathbf{var} t_1, \dots, \text{var}_n : \mathbf{var} t_n \} \subseteq \Pi} \\ \Pi \vdash e : \mathbf{exp} c' \quad c' \preceq c} \\ \Pi \vdash \mathbf{event} p \{ e \} : \mathbf{exp} c$	$\text{(LOC EXP TYPE)} \\ (\text{loc} : \mathbf{var} t) \in \Pi \\ \Pi \vdash \text{loc} : \mathbf{exp} t$
$\frac{\text{(UNDER EXP TYPE)}}{\Pi \vdash e : \mathbf{exp} t} \\ \Pi \vdash \mathbf{under} e : \mathbf{exp} t$	$\frac{\text{(REGISTER EXP TYPE)}}{\Pi \vdash e : \mathbf{exp} c} \\ \Pi \vdash \mathbf{register}(e) : \mathbf{exp} c$
$\frac{\text{(PROCEED EXP TYPE)}}{\Pi \vdash e : \mathbf{exp}(\mathbf{thunk} c)} \\ \Pi \vdash \mathbf{proceed}(e) : \mathbf{exp} c$	

Figure 11. Type-checking rules for Ptolemy.

The notation $\tau' \preceq \tau$ means τ' is a subtype of τ . It is the reflexive-transitive closure of the declared subclass relationships with the added facts that \top is a supertype of all class type expressions, and that \perp is a subtype of all class type expressions. The type \perp is used as the type of exceptions. This is formalized in Figure 13.

$$\begin{aligned} \text{isClass}(t) &= (\mathbf{class} \ t \dots) \in CT \\ \text{isThunkType}(t) &= (t = \mathbf{thunk} \ c \wedge \text{isClass}(c)) \\ \text{isType}(t) &= \text{isClass}(t) \vee \text{isThunkType}(t) \end{aligned}$$

Figure 12. Auxiliary functions not in Clifton’s dissertation.

$$\begin{array}{c} \text{(BASIS)} \\ \frac{(\mathbf{class} \ c \ \mathbf{extends} \ d \{ \dots \}) \in CT}{c \preceq d} \qquad \text{(REF)} \\ \tau \preceq \tau \end{array}$$

$$\begin{array}{ccc} \text{(TRANS)} & \text{(TOP)} & \text{(BOTTOM)} \\ \frac{\tau_1 \preceq \tau_2 \quad \tau_2 \preceq \tau_3}{\tau_1 \preceq \tau_3} & \frac{\text{isClass}(c)}{c \preceq \top} & \frac{\text{isClass}(c)}{\perp \preceq c} \end{array}$$

Figure 13. Subtyping rules, adapted from [1, Figure 3.4].

1.5 Type Soundness

The proof of soundness of Ptolemy’s type system uses a standard preservation and progress argument[14]. The details are adapted from Clifton’s work [1, 2], which in turn follows Flatt *et al.*’s work [4]. 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 a proceed closure.

DEFINITION 1.1 (*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) $S(\text{loc}) = [c.F]$, (ii) $c \preceq t$, (iii) $\text{dom}(F) = \text{dom}(\text{fieldsOf}(c))$, (iv) $\text{rng}(F) \subseteq (\text{dom}(S) \cup \{\mathbf{null}\})$, and (v) for all $f \in \text{dom}(F)$, if $F(f) = \text{loc}'$, $\text{fieldsOf}(c)(f) = u$, and $S(\text{loc}') = [c'.F']$, then $c' \preceq u$
- (b) *isThunkType*(t), $t = \mathbf{thunk} \ c$, and for some H, π, e, ρ, Π , and c' such that all the following hold: (i) $S(\text{loc}) = \mathbf{proceedClosure}(H, \mathbf{pcd} \ c, \pi)(e, \rho, \Pi)$, (ii) $\Pi \vdash e : \mathbf{exp} \ c'$, (iii) $c' \preceq c$, (iv) for each $\text{var}_i \in \text{dom}(\Pi)$, if $(\text{var}_i : \mathbf{var} \ t_i) \in \Pi$ then $\rho(\text{var}_i)$ has type t_i in S , (v) for each $\text{loc}_i \in \text{dom}(\Pi)$, if $(\text{loc}_i : \mathbf{var} \ t_i) \in \Pi$ then loc_i has type t_i in S , and (vi) for each handler record h in H , h has type $\mathbf{pcd} \ c, \pi$ in S .

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

DEFINITION 1.2 (h has type $\mathbf{pcd} \ c, \pi$ in S). *Let h be the handler record $\langle \text{loc}, m, \rho \rangle$, let c be a class name, π a type environment, and S a store. Then h has type $\mathbf{pcd} \ c, \pi$ in S if and only if for some $c', F, c_2, t', n > 1, \text{var}_i, t_i$ and e : $S(\text{loc}) = [c'.F]$, $\text{methodBody}(c', m) = (c_2, c \ m(t_1 \text{var}_1, \dots, t_n \text{var}_n)\{e\})$, $\text{dom}(\rho) = \text{dom}(\pi) = \{\text{var}_2, \dots, \text{var}_n\}$, $t_1 = \mathbf{thunk} \ c$, and for each $i \in \{2, \dots, n\}$, $(\text{var}_i : \mathbf{var} \ t_i) \in \pi$ and $\rho(\text{var}_i)$ has type t_i in S .*

The key definition of consistency is thus as follows. In the definition, $\text{tenvOf}(\nu)$ is the type environment of a frame ν , and $\text{envOf}(\nu)$ 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 1.3 (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:

- $\Pi = \text{tenvOf}(\nu)$,
- for $\rho = \text{envOf}(\nu)$, and for all $(\text{var} : \mathbf{var} \ t) \in \Pi$, $\text{var} \in \text{dom}(\rho)$ and $\rho(\text{var})$ has type t in S , and
- for all $(\text{loc} : \mathbf{var} \ t) \in \Pi$, $\text{loc} \in \text{dom}(S)$ 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 proceed 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 1.4 (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 $\Pi \approx (J, S)$, $\Pi \vdash e : \mathbf{exp} \ t$, and $\langle e, J, S, A \rangle \hookrightarrow \langle e', J', S', A' \rangle$, then there is some Π' and t' such that $\Pi' \vdash e' : \mathbf{exp} \ t'$, $t' \preceq t$ and $\Pi' \approx (J', S')$.*

Proof Sketch: The proof is by cases on the definition of \hookrightarrow (see Figure 6). Assume $\Pi \approx (J, S)$, $\Pi \vdash e : \mathbf{exp} \ t$, 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 adaptations of Clifton’s proofs for MiniMAO₀ [1, Section 3.1.4]. The result for the exception cases (see Figure 8) all follow directly from the use of $\mathbf{exp} \ \perp$ as their type and the fact that the stack in the resulting configuration is empty.

The (CALL) rule is different from Clifton’s MiniMAO₀, 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 $e = \text{loc}.m(v_1, \dots, v_n)$. From the hypotheses of the (CALL) rule we have that: $[c.F] = S(\text{loc})$, $(c_2, t' \ m(t_1 \text{var}_1, \dots, t_n \text{var}_n)\{e''\}) = \text{methodBody}(c, m)$, $\rho = \{\text{var}_i \mapsto v_i \mid 1 \leq i \leq n\} \oplus (\mathbf{this} \mapsto \text{loc})$, $\Pi'' = \{\text{var}_i : \mathbf{var} \ t_i \mid 1 \leq i \leq n\} \cup \{\mathbf{this} : \mathbf{var} \ c_2\}$, and $\nu = \mathbf{lexframe} \ \rho \ \Pi''$. So in this case, $e' = \mathbf{under} \ e''$, $J' = \nu + J$, 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 $t' \preceq t$. To prove $\Pi' \approx (\nu + J, S')$ we use definition 1.3. 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 $\rho(\text{var}_i) = \text{loc}_i \neq \mathbf{null}$, 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 (EVENT) rule, suppose $e = \mathbf{event} \ p \ \{e''\}$. From the conclusion of this rule it must be that $J = \nu + J''$ for some ν and J'' . From the hypotheses of the (EVENT) rule we have that: $\rho = \text{envOf}(\nu)$, $\Pi'' = \text{tenvOf}(\nu)$, $(c \ \mathbf{evtype} \ p \ \{t_1 \ \text{var}_1, \dots, t_n \ \text{var}_n\}) \in CT$, $\rho' = \{\text{var}_i \mapsto v_i \mid \rho(\text{var}_i) = v_i\}$, $\pi = \{\text{var}_i : \mathbf{var} \ t_i \mid 1 \leq i \leq n\}$, $\text{loc} \notin \text{dom}(S)$,

$\pi' = \pi \uplus \{loc : \mathbf{var}(\mathbf{thunk} c)\}$, $\nu' = \mathbf{evframe} p \rho' \pi'$, $H = \mathbf{hbind}(\nu' + \nu + J, S, A)$, $\theta = \mathbf{pcd} c, \pi$, and $S' = S \oplus (loc \mapsto \mathbf{proceedClosure}(H, \theta) (e, \rho, \Pi'))$. So in this case $e' = \mathbf{under}(\mathbf{proceed}(loc))$ 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 $t' \preceq t$. With these choices $\Pi' \vdash e' : \mathbf{exp} t'$, using the type rules (UNDER EXP TYPE) and (PROCEED 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 1.3. 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 (PROCEED-DONE) rule, suppose $e = \mathbf{proceed}(loc)$ and $\mathbf{proceedClosure}(\bullet, \theta) (e'', \rho'', \Pi'') = S(loc)$. From the hypothesis of this rule we have $\nu = \mathbf{lexframe} \rho'' \Pi''$. 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, $\Pi \vdash \mathbf{proceed}(loc) : \mathbf{exp} t$, from the (PROCEED EXP TYPE) rule, we have that $\Pi \vdash loc : \mathbf{exp} p$, and p is an event type whose return type is t . By hypothesis, we know that $\Pi \approx (J, S)$, and hence by definition loc has type $\mathbf{thunk} t$ in S , and thus $\Pi'' \vdash e'' : \mathbf{exp} c''$, where $c'' \preceq t$. So we choose $t' = c''$, which makes $t' \preceq t$. 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 1.3. 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 , and thus these conditions hold by parts (b)(iv) and (b)(v) in definition 1.1.

For the (PROCEED-RUN) rule, suppose $e = \mathbf{proceed}(loc)$ and $\mathbf{proceedClosure}(\langle \langle loc', m, \rho \rangle + H \rangle, \theta) (e'', \rho'', \Pi'') = S(loc)$. From the hypothesis of this rule we have $[c.F] = S(loc')$, $(c_2, t'' m(t_1 var_1, \dots, t_n var_n) \{e'''\}) = \mathbf{methodBody}(c, m)$, $n \geq 1$, $\rho_3 = \{var_i \mapsto v_i \mid 2 \leq i \leq n, v_i = \rho(var_i)\}$, $loc_1 \notin \mathbf{dom}(S)$, $S' = S \oplus (loc_1 \mapsto \mathbf{proceedClosure}(H, \theta) (e'', \rho'', \Pi''))$, $\rho_4 = \rho_3 \oplus \{var_1 \mapsto loc_1\} \oplus \{\mathbf{this} \mapsto loc'\}$, $\Pi_3 = \{var_i : \mathbf{var} t_i \mid 1 \leq i \leq n\} \uplus \{\mathbf{this} : \mathbf{var} c_2\}$, and $\nu = \mathbf{lexframe} \rho_4 \Pi_3$. So in this case we have $e' = \mathbf{under} e''$, $J' = \nu + J$, and $S' = S$. To preserve consistency, we choose $\Pi' = \Pi_3$. Since by hypothesis, $\Pi \vdash \mathbf{proceed}(loc) : \mathbf{exp} t$, from the (PROCEED 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 1.1 (b)(vi), the handler record $\langle loc', m, \rho \rangle$ has type $\theta = \mathbf{pcd} t, \pi$ in S . By definition 1.2, $t'' = t$, $\pi = \{var_2 : \mathbf{var} t_2, \dots, var_n : \mathbf{var} t_n\}$, $t_1 = \mathbf{thunk} t$, and for each $i \in \{2, \dots, n\}$, $\rho(var_i)$ 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 $t'''' \preceq t''$. So we choose $t' = t'' = t$, which makes $t' \preceq t$. 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 1.3. 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 $i \in \{2, \dots, n\}$, $\rho_4(var_i) = \rho(var_i)$, and $\rho(var_i)$ has type t_i in S , which holds the same values as S' for these locations, and (3) $\rho_4(\mathbf{this}) = loc'$ which has type c in S and hence in S' , and by the way $\mathbf{methodBody}$ works, $c \preceq c_2$, which is the type of \mathbf{this} in Π_3 . The third condition is vacuous in this case. ■

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

- [1] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.
- [2] C. Clifton and G. T. Leavens. MiniMAO₁: Investigating the semantics of proceed. *Sci. Comput. Programming*, 63(3):321–374, 2006.
- [3] C. Clifton, G. T. Leavens, and J. Noble. Ownership and effects for more effective reasoning about aspects. In *ECOOP '07, To appear*.
- [4] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. Springer-Verlag, 1999.
- [5] G. Kiczales et al. An overview of AspectJ. In *ECOOP 2001*, pages 327–353. Springer-Verlag, Jun 2001.
- [6] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD 04*, pages 26–35, 2004.
- [7] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*, pages 132–146.
- [8] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, 1995.
- [9] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD 03*, pages 90–99.
- [10] G. Nadathur and D. J. Mitchell. 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*, pages 287–291, 1999.
- [11] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE 2005*, pages 59–68.
- [12] H. Rajan and K. J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE 2003*, pages 297–306, Sep. 2003.
- [13] D. A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, MA, 1994.
- [14] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994.