Event Type Polymorphism *

Rex D. Fernando Robert Dyer Hridesh Rajan

Dept. of Computer Science, Iowa State University
Ames, IA, 50011, USA
{fernanre,rdyer,hridesh}@iastate.edu

Abstract

Subtype polymorphism is an important feature available in most modern type systems which makes code reuse and specialization possible. Recent works on separation of crosscutting concerns have created event interfaces (types) to decouple subjects from handlers. Extending the notion of subtyping to these event interfaces is a logical step. In this paper, we define event type polymorphism in the context of the Ptolemy language. Ptolemy allows declaring quantified, typed events which provide an interface between subjects and handlers. We add the notion of polymorphic event types to the Ptolemy language, defining a subtype relation among event types which in turn allows for both depth and width subtyping with regard to event context. Since Ptolemy only has explicit event announcement, our semantics is simpler and easier to reason about when compared to previously defined approaches. We also give the first formally defined static semantics for polymorphic events as well as demonstrate its usefulness via examples.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory — Semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features — Inheritance

General Terms Languages, Theory

Keywords event type, polymorphism, subtype, inheritance

1. Introduction

There has been a significant recent interest in defining an interface to fully decouple crosscutting and object-oriented (OO) concerns [2, 5, 6, 9, 10]. Among them are Ptolemy [6] and implicit invocation with implicit announcement (IIIA) [9] that propose a type-based formulation. The Ptolemy language provides a notion of quantified, typed events [6]. An *event type* abstraction defines a set of abstract events in software[1]. IIIA's join point types [9] are similar.

To illustrate, consider a simple expression language with a type checker, an evaluator, and an abstract syntax tree (AST) tracer. A parser for such a language generates AST nodes and provides a visitor, `ASTVisitor`. A visitor pattern-based design would implement the type checker, evaluator, and AST tracer as subclasses of `ASTVisitor`. This would either require making three visits over an AST or implementing a multi-dispatch to all visitors in a list[2]. An alternative design may be to define an event type per AST node type, e.g. `NumVisited` and `MultVisited`, and implement a visitor `ASTAnnouncer` that would signal these events upon visiting an AST node of the corresponding type. Consumers of AST nodes would then listen to these events and respond by doing their respective action[3], e.g. a type checker would listen to a `MultVisited` event and check whether both subexpressions of the currently visited node are well-typed. A tracer would log that expression and an evaluator would compute its value.

This alternative event-based design works out nicely using event types except for a caveat. Different consumers of AST nodes that listen to AST-related events (referred to as handlers from here onward), often require different granularity of AST events. For example, for an AST tracer it may be sufficient to treat the entire set of expression-related events at once. Similarly, for a type checker it may be sufficient to treat the entire set of binary arithmetic expressions at once – checking a binary arithmetic expression involves verifying whether left and right expressions are well-typed arithmetic

* This work was supported in part by the NSF grant CCF-10-17334. We thank Mehdi Bagherzadeh for some initial discussion.

FOAL'12, March 26, 2012, Potsdam, Germany.

[1] The main difference between Ptolemy [6] and implicit invocation [3, 8] is *quantification*, the new ability to register with a set of events using the name of the event type, which improves decoupling.

[2] A third design may be to do type checking, expression evaluation, and tracing as part of a single visitor, but that would tangle these three concerns and thus also may not be desirable.

[3] Note that sequencing different handlers in phases can be achieved by controlling which handlers are active via (un)registration.

expressions. On the other hand, an evaluator needs specific information to implement the semantics of expressions.

Thus it makes sense to provide a hierarchy of events signaled by `ASTAnnouncer`, such that handlers are able to target specific AST nodes (or subtypes thereof). Such an event hierarchy would require a notion of event polymorphism to facilitate greater reuse of handlers.

In response to this need, Steimann *et al.* defined a notion of *join point type* polymorphism [9]. However, their approach allows implicit announcement, which causes the semantics of subtyping to become slightly confusing for a few cases. Thus there is a need for a simpler notion of event polymorphism.

In this paper, we propose our notion of event type polymorphism. Unlike previous work, our approach is based on Ptolemy which has only explicit event announcement. The lack of implicit announcement makes our semantics of event type polymorphism simpler and allows for easier reasoning.

In the rest of the paper, we describe our initial formalization and static semantics of event type polymorphism in an extension of the Ptolemy language we call *Ptolemy*$_{\ll:}$.

2. Polymorphic Event Types

In this section we present *Ptolemy*$_{\ll:}$, a language model that enhances Ptolemy [6] with the notion of polymorphic event types.

2.1 Abstract Syntax

The syntax is presented in Figure 1. This syntax defines a program as a collection of declarations followed by an expression, which is like a main method in Java. There are two kinds of declarations: events and classes. Classes are defined with a single inheritance model. In examples and in concrete syntax, we take an omitted **`extends`** clause to mean **`extends`** `Object`. For simplicity, this syntax does not provide packages, modifiers, abstract classes and methods, static members, interfaces, constructors, initializers, or built-in types. Ptolemy has a unified language model like Eos [7], i.e. handlers are just normal classes with regular methods inside that take a handler chain as first argument. A class may have zero or more fields, methods, and binding declarations.

An event type declaration is a new feature in Ptolemy, which gives a name to a set of abstract events. A novelty of *Ptolemy*$_{\ll:}$ is to add a single event inheritance model to event declarations. An event type may extend exactly one event type or the top event type `Event`. In examples and in concrete syntax, we take an omitted **`extends`** clause to mean **`extends`** `Event`.

A binding declaration in Ptolemy is the main quantification feature. It declares an event of interest for a handler and corresponding handler method. For example, a binding declaration **`when`** `p` **`do`** `m;` says to run method `m` when events of *exact* type `p` are announced. This semantics is modified. In *Ptolemy*$_{\ll:}$, a binding declaration **`when`** `p` **`do`** `m;` says to run method `m` when events of type `p` or *any of its subevents* are announced. This allows quantification over event types in an event inheritance hierarchy.

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

where

c	$\in$	$\mathcal{C}$, a set of class names
d	$\in$	$\mathcal{C} \cup \{Object\}$, a set of superclass names
p	$\in$	$\mathcal{P}$, a set of event type names
q	$\in$	$\mathcal{P} \cup \{Event\}$, a set of super event type names
f	$\in$	$\mathcal{F}$, a set of field names
m	$\in$	$\mathcal{M}$, a set of method names
var	$\in$	{**this**} $\cup \mathcal{V}$, $\mathcal{V}$ is a set of variable names

Figure 1. *Ptolemy*$_{\ll:}$'s syntax[6] with polymorphic events

There are standard OO expressions in the syntax for object constructions, *var*, **`null`**, method call, field get and set, cast, and local variable definition.

Event-related expressions are for (un)registration, announcement, and for running the event handler chain. The registration expression **`register`**`(e)` evaluates `e` to a location and then takes all binding declarations in the declaring class of the object stored at that location and makes them active. The expression **`unregister`**`(e)` naturally deactivates such bindings. An announce expression **`announce`** `p(e1,..,en){e}` in *Ptolemy*$_{\ll:}$ evaluates `e1,..,en` to values `v1,..,vn`, creates a closure for expression `e`, binds `v1,..,vn` to context variables of `p` in the newly created closure, creates a chain of registered handlers for `p` and all its superevents (up to the top event type `Event`) and runs the first handler in that chain. The handlers in this chain are in the order of their dynamic registration. The last link in this chain is the previously created event closure. These handlers may access a subset of values `v1,..,vn` via their first parameter, the event handler chain. The invoke expression `e.`**`invoke`**`()` evaluates `e`, which must evaluate to an event handler chain and runs the next element in this chain.

2.2 Events in Expression Interpreter

An example in Ptolemy is given in Figure 2. On the left side, we see several declarations of events with no notion of event type polymorphism. Notice the repeated `left` and `right` context declarations (left side, lines 5, 10, and 15). On the right side, we give the same event type declarations but with inheritance. You can see both depth subtyping [1] (the context `node` is narrowed the further down the inheritance tree you go, e.g. right side, lines 2, 4, 9, and 12) as well as width subtyping (new contexts `left`/`right` appear in event `BinaryArithVisited` right side, line 5).

These events are then used by event handlers, such as the one in Figure 3. Again we show both versions without and

WITHOUT EVENT SUBTYPING

```
1  void event ExpVisited { Exp node; }
2  void event ArithVisited { Arith node; }
3  void event BinaryArithVisited {
4    BinaryArith node;
5    Exp left;
6    Exp right;
7  }
8  void event MultVisited {
9    Mult node;
10   Exp left;
11   Exp right;
12 }
13 void event DivVisited {
14   Div node;
15   Exp left;
16   Exp right;
17 }
```

WITH EVENT SUBTYPING

```
1  void event ExpVisited { Exp node; }
2  void event ArithVisited extends ExpVisited { Arith node; }
3  void event BinaryArithVisited extends ArithVisited {
4    BinaryArith node;
5    Exp left;
6    Exp right;
7  }
8  void event MultVisited extends BinaryArithVisited {
9    Mult node;
10 }
11 void event DivVisited extends BinaryArithVisited {
12   Div node;
13 }
```

Figure 2. Event hierarchy for an expression visitor. These events are announced by the visitor as it parses nodes of that type.

WITHOUT EVENT SUBTYPING

```
1  class ASTTracer {
2    void printArith(ArithVisited next){
3      logVisitBegin(next.class);
4      next.invoke();
5      logVisitEnd(next.class);
6    } when ArithVisited do printArith;

8    void printBArith(BinaryArithVisited next){
9      logVisitBegin(next.class);
10     next.invoke();
11     logVisitEnd(next.class);
12   } when BinaryArithVisited do printBArith;

14   void printMult(MultVisited next){
15     logVisitBegin(next.class);
16     next.invoke();
17     logVisitEnd(next.class);
18   } when MultVisited do printMult;

20   void printDiv(DivVisited next){
21     logVisitBegin(next.class);
22     next.invoke();
23     logVisitEnd(next.class);
24   } when DivVisited do printDiv;
25 }
```

Figure 3. Expression AST tracing.

WITH EVENT SUBTYPING

```
1  class ASTTracer {
2    void printExp(ExpVisited next) {
3      logVisitBegin(next.node().class);
4      next.invoke();
5      logVisitEnd(next.node().class);
6    } when ExpVisited do printExp;
7  }
```

Figure 4. Expression AST tracing. Note significant reuse of handler code due to event subtyping.

```
1  class Checker {
2    // code for interface Type elided
3    static class NumType implements Type {}
4    Stack typeStack = new Stack();

6    void chkBinArth (BinaryArithVisited next){
7      next.invoke();
8      NumType t1 = (NumType) typeStack.pop();
9      NumType t2 = (NumType) typeStack.pop();
10     typeStack.push(new NumType());
11   } when BinaryArithVisited do chkBinArth;
12 }
```

Figure 5. Expression type checking

with the notion of event type polymorphism in Figure 3 and Figure 4 respectively. As you can see, adding event type polymorphism allows targeting a large number of events with only one handler (Figure 4, lines 2–6), thus providing a higher degree of code reuse in the system.

Examples of the new syntax are shown in the right side of Figure 2 and in Figure 4. An additional example that demonstrates type checking for the expression language is shown in Figure 5. Note that once again, event type polymorphism has allowed us to only create one event handler (for the event `BinaryArithExpVisited`, lines 6–11). Without this new feature, the class would require 2 handlers with identical bodies. This trend also extends to the other AST types (not shown here), allowing the full type checking class to save around 50 lines of code (33%) and have half as many handlers.

3. Static Semantics

In this section, we give the static semantics of *Ptolemy*$_{\ll:}$. The semantics uses the attributes defined in Figure 6. These attributes are borrowed from [6][4].

We state the type checking rules using a fixed class table (list of declarations CT). The class table can be thought of as an implicit inherited attribute used by the rules and auxiliary

[4] The only exception is that the original paper [6, Fig. 7] had types for pointcut descriptions (*pcd*), which were later eliminated.

(IS EVENT)
$$\frac{(c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT}{\mathit{isEvent}(p)}$$

(≪: TOP)
$$\frac{\mathit{isEvent}(p)}{p \ll: \mathit{Event}}$$

(≪: TRANS.)
$$\frac{\mathit{isEvent}(p) \quad \mathit{isEvent}(q) \quad \mathit{isEvent}(q') \quad p \ll: q' \quad q' \ll: q}{p \ll: q}$$

(≪: REFL.)
$$\frac{\mathit{isEvent}(p)}{p \ll: p}$$

(≪: BASE)
$$\frac{(c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT \quad \mathit{isEvent}(q) \quad [t'_1\ var'_1, ..., t'_m\ var'_m] = \mathit{contextsOf}(q) \quad (\forall i \in [1..n] :: t_i\ var_i \in [t_1\ var_1, ..., t_n\ var_n] \Rightarrow (\exists j \in [1..m] :: t'_j\ var_i \in [t'_1\ var'_1, ..., t'_m\ var'_m])) \Rightarrow t_i <: t'_j}{p \ll: q}$$

Figure 7. Definition of sub-event type relation ≪:

$\theta ::=$	"type attributes"
OK	"program/top-level decl."
| OK in c	"method, binding"
| **var** t	"var/formal/field"
| **exp** t	"expression"
$\tau ::= c \mid \top \mid \bot$	"class type expressions"
$\pi, \Pi ::= \{I : \theta_I\}_{I \in K}$, where K is finite, $K \subseteq (\mathcal{L} \cup \{\texttt{this}\} \cup \mathcal{V})$	"type environments"

Figure 6. Type attributes (based on [6, Fig. 7])

functions. We require that top-level names in the program are distinct and that the inheritance relation on classes **and events** is acyclic.

These rules make use of auxiliary relations *concreteType* and *contextsOf* to compute all context variables (inherited and current) of an event type. The relation *concreteType* is defined by the two rules (CONCRETE TYPE INH.) and (CONCRETE TYPE DEPTH) in Figure 8. This relation takes a context declaration and a list of context declarations and returns the former if no context with the same name is in the list, otherwise it returns the context type from the list. The relation *contextsOf* is defined by the two rules (CONTEXT VARS) and (TOP CONTEXT VARS) in Figure 8. This relation walks the declared event inheritance hierarchy and collects concrete context variable declarations.

(CONCRETE TYPE INH.)
$$\frac{var'_i \notin \{var_1, ..., var_n\}}{\mathit{concreteType}(t'_i\ var'_i, [t_1\ var_1, ..., t_n\ var_n]) = t'_i\ var'_i}$$

(CONCRETE TYPE DEPTH)
$$\frac{\exists j \in [1..n] :: t_j\ var'_i \in [t_1\ var_1, ..., t_n\ var_n]}{\mathit{concreteType}(t'_i\ var'_i, [t_1\ var_1, ..., t_n\ var_n]) = t_j\ var'_i}$$

(TOP CONTEXT VARS)
$$\frac{}{\mathit{contextsOf}(Event) = \bullet}$$

(CONTEXT VARS)
$$\frac{(c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT \quad [t'_1\ var'_1, ..., t'_m\ var'_m] = \mathit{contextsOf}(q)}{\mathit{contextsOf}(p) = [\forall i \in [1..m] :: \mathit{concreteType}(t'_i\ var'_i, [t_1\ var_1, ..., t_n\ var_n])] + [\forall i \in [1..n] :: t_i\ var_i :: var_i \notin \{var'_1, ..., var'_m\})]}$$

Figure 8. Definition of *contextsOf* relation

The main new relation in the type system formalizes event subtyping. We write $p \ll: q$ to mean that p is a well-typed subevent of q. This relation is defined in Figure 7. The definition of this relation makes uses of one other relation: *isEvent*. The relation *isEvent* is defined by the rule (IS EVENT) that checks the program's class table CT for that event's presence.

The rule (≪: TOP) defines the event type `Event` as the root of the subtyping relation. The rules (≪: TRANS) and (≪: REFL) define event subtyping as a transitive and reflexive relation.

The rule (≪: BASE) defines the depth [1] and width subtyping for event inheritance. This rule allows subevent types to specialize the type of a context variable. An example usage was presented in Figure 2, where the context variable `node`'s type `Exp` in the event `ExpVisited` was specialized by subevent `ArithVisited` to be `Arith`. This context variable type was further specialized by the subevent `BinaryArithVisited` to be `BinaryArith`. Allowing specialization thus permits passing richer context about an event without an increased number of context variables.

The rule also indirectly defines the width subtyping for event inheritance, via the *contextsOf* relation. This rule allows subevent types to add additional context variables. An example usage was presented in Figure 2, where the context variables `left` and `right` were added to the subevent `BinaryArithVisited`.

Figure 9 presents the type rules which are modified from [6]. All omitted rules are essentially unmodified.

(CHECK EVENT)
$$\frac{\mathit{isClass}(c) \quad \forall i \in [1..n] :: \mathit{isClass}(t_i) \quad p \ll: q}{\Pi \vdash c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\} : \text{OK}}$$

(CHECK BINDING)
$$\frac{\mathit{isClass}(c') \quad (c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT \quad c' <: c \quad (c'\ m\ (\texttt{thunk}\ p\ var)\ \{e\}) = \mathit{methodBody}(c, m)}{\Pi \vdash \texttt{when}\ p\ \texttt{do}\ m : \text{OK in } c}$$

(ANNOUNCE EXP TYPE)
$$\frac{\mathit{isEvent}(p) \quad [t_1\ var_1, ..., t_n\ var_n] = \mathit{contextsOf}(p) \quad \Pi \vdash e_1 : \texttt{exp}\ t_1\ \ldots\ \Pi \vdash e_n : \texttt{exp}\ t_n \quad \Pi \vdash e : \texttt{exp}\ c' \quad c' <: c}{\Pi \vdash \texttt{announce}\ p\ (e_1, \ldots, e_n)\ \{e\} : \texttt{exp}\ c}$$

Figure 9. Type-checking rules (based on [6, Fig. 8])

The (CHECK EVENT) rule makes use of the relation $p \ll: q$ defined in Figure 7. This rule also verifies that all declared types of context variables and the return type of the event c are valid class types (event type names are not allowed as context variable types).

The (CHECK BINDING) rule in [6] specifies that the handler method should take context variables of the announced event as arguments. As we have changed that behavior as stated previously, modifications have been made to this rule to reflect those changes. Handler methods should only take one argument, the event handler chain that has type (**thunk** p).

The (ANNOUNCE EXP TYPE) rule (based on rule (EVENT) [6]) is modified in order for the `announce` expression to take context variable values as arguments instead of collecting them from the surrounding environment. Also, an event's context variables are now defined as its immediately declared context variables as well as any context variables which are declared in a superevent. $contextsOf(p)$ calculates that list of context variables, in the order they were declared (superevent context variables are first, new context variables are last to handle width subtyping) and with the concrete type (handling depth subtyping)

4. Related Work

Similar to Ptolemy [6], Steimann *et al.* describe a system with implicit invocation and implicit announcement (IIIA) that contains a notion of an event interface between base code and aspects [9]. Their notion of a *join point type* is analagous to Ptolemy's notion of *event type* in that both provide named types to describe events and any context information passed by those events. Since their system also allows implicit announcement (unlike Ptolemy's explicit only announcement) their join point types also provide pointcuts. Join point types may define a subtyping relation, with semantics that are very similar to those described in this paper. Join point subtypes inherit the context from their supertypes and have the ability to specify additional context.

The key differences between $Ptolemy_{\ll:}$ and IIIA relate to IIIA's support for implicit announcement, which complicates the semantics and makes reasoning about the code difficult. Since IIIA allows implicit announcement, the semantics of subtyping become slightly confusing in a few cases. For example consider a class that exhibits two join point types and the pointcuts of both types match the same point of execution. The question becomes, how many event announcements occur at that location? The answer depends on the relationship between the two join point types. If one subtypes the other, then only the most specific type is announced at that point. In all other cases, two events will be announced at that location. However note that if the two types are siblings with a common supertype, a handler listening to the supertype will now execute twice. In the case where one type subtyped the other, there is only one event announcement and thus the handler listening to the supertype only executes once. $Ptolemy_{\ll:}$ does not suffer from this because it allows only explicit announcement.

Gasiunas *et al.* extend the Scala language with explicitly defined events, which can be imperative or declarative [4]. Events are declared as members in classes and are inherited in the classes' subtypes. The inheritance semantics is analogous to method inheritance, which restricts the event subtyping to depth subtyping.

5. Conclusion and Future Work

Several recent languages provide a new interface to decouple subjects from handlers [6, 9]. Extending the notion of subtype polymorphism to such interfaces seems like a logical next step. In this paper we showed examples to demonstrate the level of code reuse possible with event type polymorphism, clearly demonstrating its benefit. Prior work provided a notion of event polymorphism [9], however their semantics are complicated due to the language allowing implicit announcement. We provide a simpler semantics for the Ptolemy language (which only allows explicit announcement) and gave our initial formalization of those semantics.

In the future, we plan to finish formalizing these semantics as well as prove the soundness of the type system. We have writen out the static and dynamic semantics rules in Coq [11]. A type soundness proof using the standard progress and preservation argument [12] is in progress. We also plan to finish an implemention of these semantics in the OpenJDK-based Ptolemy compiler.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] J. Aldrich. Open Modules: Modular reasoning about advice. In *ECOOP*, pages 144–168, 2005.

[3] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM*, pages 31–44, 1991.

[4] V. Gasiunas, L. Satabin, M. Mezini, A. N. nez, and J. Noyé. Escala: modular event-driven object interactions in scala. In *AOSD*, pages 227–240. ACM, 2011.

[5] K. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ '07*, pages 63–72, 2007.

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

[7] H. Rajan and K. J. Sullivan. Unifying aspect- and object-oriented design. *ACM TOSEM*, 19(1), August 2009.

[8] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Softw.*, 7(4):57–66, 1990.

[9] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM TOSEM*, 20:1–43, July 2010.

[10] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *ACM TOSEM*, 20(2), 2011.

[11] The Coq Development Team. The coq proof assistant reference manual. Technical Report V8.3pl2, INRIA, 2011.

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