Event Type Polymorphism

Rex D. Fernando        Robert Dyer        Hridesh Rajan

Department of Computer Science
Iowa State University
{fernanre,rdyer,hridesh}@iastate.edu

This work was supported in part by NSF grant CCF-10-17334.
Motivation: Code re-use and specialization for event-based separation of concerns

Approach: Event Type Polymorphism in Ptolemy

Technical Contributions:
- Formal semantics for event type polymorphism
- Simpler semantics, when compared to earlier work
An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

AST

MultExp

DivExp

NumExp

NumExp

NumExp
An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

class C announces event E

Rex D. Fernando, Robert Dyer and Hridesh Rajan
**Overview**

**Motivation**

**Language**

**Summary**

An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

---

**AST**

```
    MultExp
     /   \
DivExp  NumExp
     /   \
NumExp  NumExp
```

**Events**

```
    MultVisited
     /   \
DivVisited
```

---

C  E  

*class C announces event E*
An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

class C announces event E
class C announces event E

event E's announcement invokes handler H
An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

---

**Overview**

**Motivation**

**Language**

**Summary**

---

**NumExp**

**DivExp**

**MultExp**

**NumVisited**

**DivVisited**

**Tracing**

**TypeCheck**

**Evaluation**

---

**C**

class C announces event E

**E**

event E's announcement invokes handler H

---

Rex D. Fernando, Robert Dyer and Hridesh Rajan

---

Event Type Polymorphism
class C announces event E

event E's announcement invokes handler H
### MultVisited

<table>
<thead>
<tr>
<th>Event</th>
<th>Node: MultExp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left: Exp</td>
</tr>
<tr>
<td></td>
<td>right: Exp</td>
</tr>
</tbody>
</table>

### DivVisited

<table>
<thead>
<tr>
<th>Event</th>
<th>Node: DivExp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left: Exp</td>
</tr>
<tr>
<td></td>
<td>right: Exp</td>
</tr>
</tbody>
</table>

### PlusVisited

<table>
<thead>
<tr>
<th>Event</th>
<th>Node: PlusExp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left: Exp</td>
</tr>
<tr>
<td></td>
<td>right: Exp</td>
</tr>
</tbody>
</table>
### Overview

**Motivation**

**Language**

**Summary**

---

#### An Example - No Event Type Polymorphism

1. **MultVisited**
   - `<event>`
   - `+ node : MultExp`
   - `+ left : Exp`
   - `+ right : Exp`

2. **DivVisited**
   - `<event>`
   - `+ node : DivExp`
   - `+ left : Exp`
   - `+ right : Exp`

3. **PlusVisited**
   - `<event>`
   - `+ node : PlusExp`
   - `+ left : Exp`
   - `+ right : Exp`

---

Rex D. Fernando, Robert Dyer and Hridesh Rajan

---

Event Type Polymorphism
<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| MultVisited | + node: MultExp  
+ left: Exp  
+ right: Exp |
| DivVisited | + node: DivExp  
+ left: Exp  
+ right: Exp |
| PlusVisited | + node: PlusExp  
+ left: Exp  
+ right: Exp |
class ASTTracer {
    void printMult(MultVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    } when MultVisited do printMult;

    void printDiv(DivVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    } when DivVisited do printDiv;

    void printPlus(PlusVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    } when PlusVisited do printPlus;
}
```java
class ASTTracer {
    void printMult(MultVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    } when MultVisited do printMult;

    void printDiv(DivVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    } when DivVisited do printDiv;

    void printPlus(PlusVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    } when PlusVisited do printPlus;
}
```
Can we re-use code here?

What happens if a new AST type is added?

What happens if an AST type is removed?
Can we re-use code here?

- **No!** Passing event closures (next) as argument is illegal.
  (to simplify reasoning about invoke/proceed functionality)

What happens if a new AST type is added?

What happens if an AST type is removed?
Can we re-use code here?

- **No!** Passing event closures (`next`) as argument is illegal. (to simplify reasoning about `invoke/proceed` functionality)

What happens if a new AST type is added?

- Must update all handlers to support that node type!

What happens if an AST type is removed?
Can we re-use code here?
   Yes! Passing event closures (next) as argument is illegal.
   (to simplify reasoning about invoke/proceed functionality)

What happens if a new AST type is added?
   Must update all handlers to support that node type!

What happens if an AST type is removed?
   Must update all handlers and remove that node type!
Can we re-use code here?

- **No!** Passing event closures (`next`) as argument is illegal. (to simplify reasoning about `invoke/proceed` functionality)

What happens if a new AST type is added?
- Must update **all** handlers to support that node type!

What happens if an AST type is removed?
- Must update **all** handlers and remove that node type!

**Polymorphism can help us here!**
Overview

Motivation

Language

Summary

An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

MultVisited

<event>

+ node : MultExp

DivVisited

<event>

+ node : DivExp

PlusVisited

<event>

+ node : PlusExp

ExpVisited

<event>

+ node : Exp

BinArithVisited

<event>

+ node : BinArith

+ left : Exp

+ right : Exp

Rex D. Fernando, Robert Dyer and Hridesh Rajan
Overview

Motivation

Language

Summary

An Example - No Event Type Polymorphism

Example Revisited - With Event Type Polymorphism

MultVisited

<event>
+ node : MultExp

DivVisited

<event>
+ node : DivExp

PlusVisited

<event>
+ node : PlusExp

ExpVisited

<event>
+ node : Exp

BinArithVisited

<event>
+ node : BinArith
  + left : Exp
  + right : Exp

Rex D. Fernando, Robert Dyer and Hridesh Rajan

Event Type Polymorphism
class ASTTracer {
    void printExp (ExpVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    }
    when ExpVisited do printExp;
}
class ASTTracer {
    void printExp(ExpVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    }
    when ExpVisited do printExp;
}

▶ Quantifying over entire event hierarchy by only naming super event!
▶ No need to update when a new AST type added!
▶ No need to update when an AST type removed!
class ASTTracer {
    void printExp(ExpVisited next) {
        logVisitBegin(next.node().getClass());
        next.invoke();
        logVisitEnd(next.node().getClass());
    }
    when ExpVisited do printExp;
}

- Quantifying over entire event hierarchy by only naming super event!
- No need to update when a new AST type added!
- No need to update when an AST type removed!

Let's take a look at the language...
\[ \text{decl} ::= \text{class } c \text{ extends } d \{ \text{ field* meth* binding* } \} \]
\[ \text{ | } c \text{ event } p \text{ extends } q \{ \text{ form* } \} \]

where
\[ c \in C, \text{ a set of class names} \]
\[ d \in C \cup \{ \text{Object} \}, \text{ a set of superclass names} \]
\[ p \in P, \text{ a set of event type names} \]
\[ q \in P \cup \{ \text{Event} \}, \text{ a set of super event type names} \]
binding ::= when p do m

e ::= register(e) | unregister(e) 
| announce p (e*) { e } 
| e.invoke()

where

m ∈ M, a set of method names
(CHECK EVENT)

\[
\text{isClass}(c) \quad \forall i \in [1..n] :: \text{isClass}(t_i) \quad \text{p} \ll : \text{q}
\]

\[\prod \vdash c \text{ event p extends q } \{ t_1 \text{ var}_1, \ldots, t_n \text{ var}_n \} : \text{OK}\]
(≪: Top)
\[
\begin{align*}
isEvent(p) \\
p ≪: Event
\end{align*}
\]

(≪: Reflexive)
\[
\begin{align*}
isEvent(p) \\
p ≪: p
\end{align*}
\]

(≪: Transitive)
\[
\begin{align*}
isEvent(p) \\
isEvent(q) & \quad isEvent(q') \\
p ≪: q' & \quad q' ≪: q \\
p ≪: q
\end{align*}
\]
(\ll: \textsc{Base})

(c \textup{ event } p \ \textup{extends} \ q \ \{t_1 \ \textit{var}_1, \ldots, t_n \ \textit{var}_n\}) \in CT

\text{isEvent}(q) \quad [t'_1 \ \textit{var}'_1, \ldots, t'_m \ \textit{var}'_m] = \text{contextsOf}(q)

\forall i \in [1..n] :: t_i \ \textit{var}_i \in [t_1 \ \textit{var}_1, \ldots, t_n \ \textit{var}_n] \Rightarrow

(\exists j \in [1..m] :: t'_j \ \textit{var}_i \in [t'_1 \ \textit{var}'_1, \ldots, t'_m \ \textit{var}'_m] \Rightarrow t_i <: t'_j)

p \ \ll: \ q

\text{contextsOf} \ \text{recursively computes the list of all context for an event type } q, \ \text{based on its supertypes}
- New syntax: \( p \) extends \( q \)
- Typing rules use new relation: \( p \ll q \)
- Both depth and width subtyping of context information
Related Work

- Implicit Invocation + Implicit Announcement [Steimann 2010]
  - Implicit announcement allows ambiguity
  - Harder to reason about what event(s) announced

- Escala [Gasiunas 2011]
  - Does not support width subtyping
  - Limits the ability to specialize sub-events
Future Work

- Finish type-soundness proof (in Coq)
- Implement semantics in OpenJDK-based Ptolemy compiler
  - Non-trivial to implement
Motivation: Code re-use and specialization for event-based separation of concerns

- Ability to quantify over a hierarchy of events
- Allows for code re-use in event definitions and handlers
- Better maintenance - for both adding and removing events

Approach: Event Type Polymorphism in Ptolemy

- Event types have inheritance
- Allow width and depth subtyping of context
- Handlers also handle sub-events

Technical Contributions:

- Formal semantics for event type polymorphism
- Simpler semantics, when compared to earlier work
Questions?

http://ptolemy.cs.iastate.edu/
Appendix

Full Syntax
Auxilliary Methods
Type-checking Rules
Future Work

\[ prog ::= decl^* e \]

\[ decl ::= \text{class } c \text{ extends } d \{ \text{field}^* \text{ meth}^* \text{ binding}^* \} \]
\[ | \text{c event } p \text{ extends } q \{ \text{form}^* \} \]

where
\[
c \in C, \text{ a set of class names}
d \in C \cup \{ \text{Object} \}, \text{ a set of superclass names}
\]
\[
p \in P, \text{ a set of event type names}
q \in P \cup \{ \text{Event} \}, \text{ a set of super event type names}
\]
\[ t ::= c \mid \text{thunk } p \]

\[ \text{field} ::= c \ f \]
\[ \text{meth} ::= c \ m \ (\text{form}^*) \{ e \} \]
\[ \text{form} ::= t \ \text{var}, \quad \text{where } \text{var} \neq \text{this} \]

\[ \text{binding} ::= \text{when } p \ \text{do } m \]

**where**

\[ f \in \mathcal{F}, \text{ a set of field names} \]
\[ m \in \mathcal{M}, \text{ a set of method names} \]
\[ \text{var} \in \{\text{this}\} \cup \mathcal{V}, \mathcal{V} \text{ is a set of variable names} \]
Appendix

Full Syntax

Auxilliary Methods

Type-checking Rules

Future Work

\[
ep ::= n \mid \text{var} \mid ep.f \mid ep \neq \text{null} \mid ep == ep \\
    \mid ep < ep \mid ! ep \mid ep \&\& ep
\]

\[
e ::= \text{new } c() \mid \text{var} \mid \text{null} \mid e.m(e^*) \mid e.f \\
    \mid e.f = e \mid \text{cast } c \ e \mid \text{form } = e ; e \mid e ; e \\
    \mid \text{if } (ep) \{ e \} \text{ else } \{ e \} \mid \text{while } (ep) \{ e \} \\
    \mid \text{register}(e) \mid \text{unregister}(e) \\
    \mid \text{announce } p \ (e^*) \{ e \} \\
    \mid e.\text{invoke}()
\]

where
\[n ∈ \mathbb{Z}, \text{ the set of integers}\]
(Concrete Type Inh.)

\[ \text{var}'_i \not\in \{\text{var}_1, \ldots, \text{var}_n\} \]

\[ \text{concreteType}(t'_i \text{ var}'_i, [t_1 \text{ var}_1, \ldots, t_n \text{ var}_n]) = t'_i \text{ var}'_i \]

(Concrete Type Depth)

\[ \exists j \in [1..n] :: t_j \text{ var}'_i \in [t_1 \text{ var}_1, \ldots, t_n \text{ var}_n] \]

\[ \text{concreteType}(t'_i \text{ var}'_i, [t_1 \text{ var}_1, \ldots, t_n \text{ var}_n]) = t_j \text{ var}'_i \]
\[(\text{Top Context Vars})\]
\[
\text{contextsOf}(\text{Event}) = \bullet
\]

\[(\text{Context Vars})\]
\[
\begin{align*}
\text{(c event } p \text{ extends } q \{t_1 \; \text{var}_1, \ldots, t_n \; \text{var}_n\}) & \in CT \\
[t'_1 \; \text{var}'_1, \ldots, t'_m \; \text{var}'_m] & = \text{contextsOf}(q) \\
\text{contextsOf}(p) & = \\
[\forall i \in [1..m] : : \text{concreteType}(t'_i \; \text{var}'_i, [t_1 \; \text{var}_1, \ldots, t_n \; \text{var}_n])] \\
+ [\forall i \in [1..n] : : t_i \; \text{var}_i : : \text{var}_i \not\in \{\text{var}'_1, \ldots, \text{var}'_m\}] 
\end{align*}
\]
\[
\text{contextsOf(ExpVisited)} = [\text{node: Exp}]
\]
\[
\text{contextsOf(BinArithVisited)} = [\text{node: BinArith, left: Exp, right: Exp}]
\]
\[
\text{contextsOf(DivVisited)} = [\text{node: DivExp, left: Exp, right: Exp}]
\]
(Is Event)
\[(c \text{ event } p \text{ extends } q \{t_1 \ var_1, \ldots, t_n \ var_n\}) \in CT \]

isEvent\( (p) \)
(≪: Top) \[
\text{isEvent}(p) \\
p ≪: \text{Event}
\]

(≪: Refl.) \[
\text{isEvent}(p) \\
p ≪: p
\]

(≪: Trans.) \[
isEvent(p) \\
isEvent(q) \quad isEvent(q') \\
p ≪: q' \quad q' ≪: q
\]
\[
p ≪: q
\]
(\llangle \bowtie: \text{BASE} \rangle)

(c \text{ event } p \text{ extends } q \{t_1 \text{ var}_1, \ldots, t_n \text{ var}_n\}) \in CT

\text{isEvent}(q) \quad [t'_1 \text{ var}'_1, \ldots, t'_m \text{ var}'_m] = \text{contextsOf}(q)

\forall i \in [1..n] :: t_i \text{ var}_i \in [t_1 \text{ var}_1, \ldots, t_n \text{ var}_n] \Rightarrow

(\exists j \in [1..m] :: t'_j \text{ var}_i \in [t'_1 \text{ var}'_1, \ldots, t'_m \text{ var}'_m] \Rightarrow t_i \ll t'_j)

p \ll q
\[ \theta ::= \]
\[ \quad \text{OK} \quad \text{“program/top-level declaration”} \]
\[ \quad \text{OK in } c \quad \text{“method, binding”} \]
\[ \quad \text{var } t \quad \text{“var/formal/field”} \]
\[ \quad \text{exp } t \quad \text{“expression”} \]

\[ \tau ::= c \mid \top \mid \bot \quad \text{“class type expressions”} \]

\[ \pi, \Pi ::= \{ I : \theta_I \}_{I \in K}, \quad \text{“type environments”} \]
\[ \quad \text{where } K \text{ is finite, } K \subseteq (\mathcal{L} \cup \{ \text{this} \} \cup \mathcal{V}) \]
(CHECK EVENT)

\[ \text{isClass}(c) \quad \forall i \in [1..n] :: \text{isClass}(t_i) \]

\[ \prod \vdash c \text{ event } p \text{ extends } q \{ t_1 \ var_1, \ldots, t_n \ var_n \} : \text{OK} \]
(CHECK BINDING)

\[
\text{isClass}(c') \\
(c \text{ event } p \text{ extends } q \{ t_1 \text{ var}_1, \ldots, t_n \text{ var}_n \}) \in CT \\
c' <: c \quad (c' \ m(\text{thunk} \ p \ \text{var})\{e\}) = \text{methodBody}(c, m) \\
\Pi \vdash \text{when } p \text{ do } m : \text{OK} \text{ in } c
\]
(ANNOUNCE EXP TYPE)

\[
(c \text{ event } p \text{ extends } q \{t_1 \text{ var}_1, \ldots, t_n \text{ var}_n\}) \in CT
\]

\[
\forall i \in [1..n] :: \Pi \vdash e_i : \text{exp } t_i \\
\Pi \vdash e : \text{exp } c' \\
c' \prec c
\]

\[
\Pi \vdash \text{announce } p(e_1, \ldots, e_n) \{e\} : \text{exp } c
\]
Implementation

- Static semantics are relatively simple

- But implementation is non-trivial
  - Handling a supertype event requires the entire hierarchy rooted by that event also be registered
  - But to maintain separate compilation and type checking, event types are only aware of their direct supertype
  - What happens when loading new subtypes and handlers already registered?