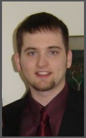
Event Type Polymorphism

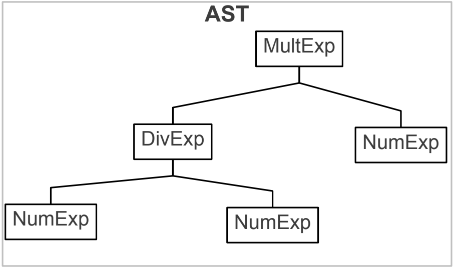
Rex D. Fernando

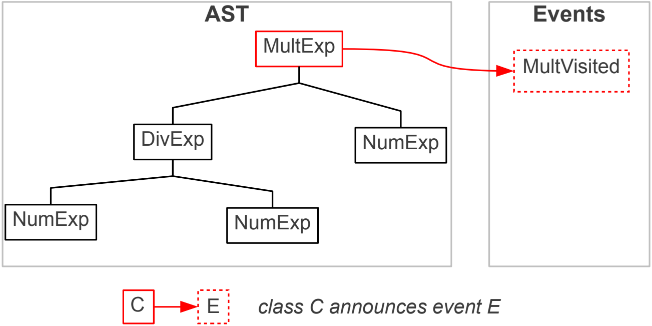
Robert Dyer

Hridesh Rajan


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

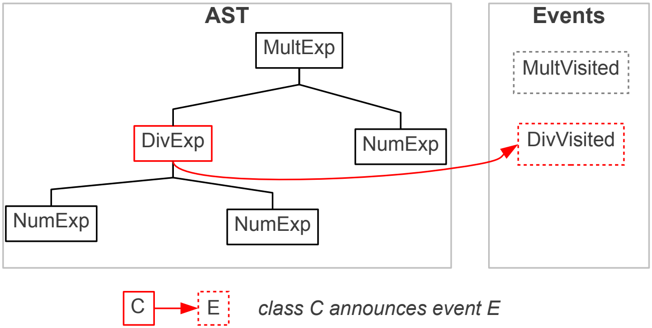
This work was supported in part by NSF grant CCF-10-17334.

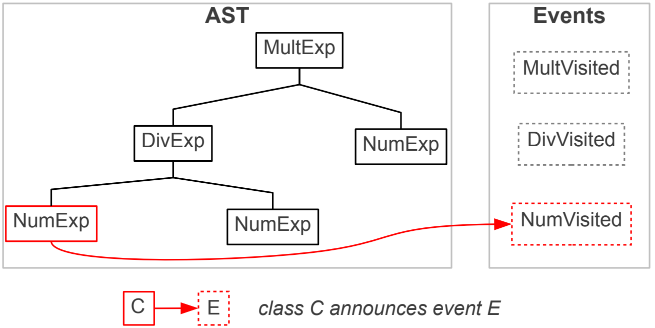
- Motivation: Code re-use and specialization for event-based separation of concerns
- Approach: Event Type Polymorphism in Ptolemy
- Technical Contributions:
  - Formal semantics for event type polymorphism
  - Simpler semantics, when compared to earlier work

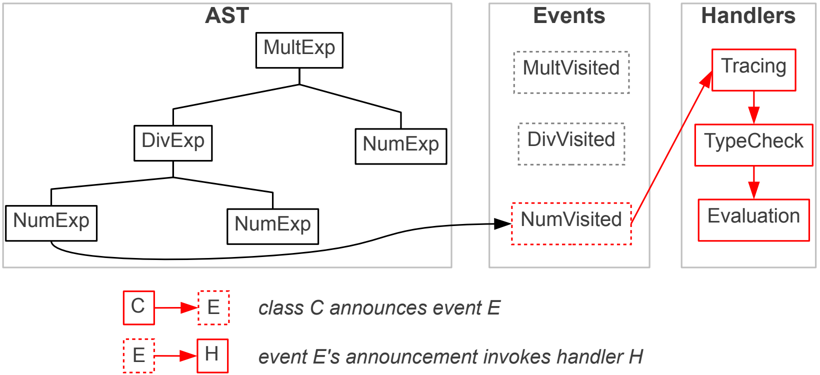

AST
MultExp
DivExp
NumExp
NumExp
NumExp

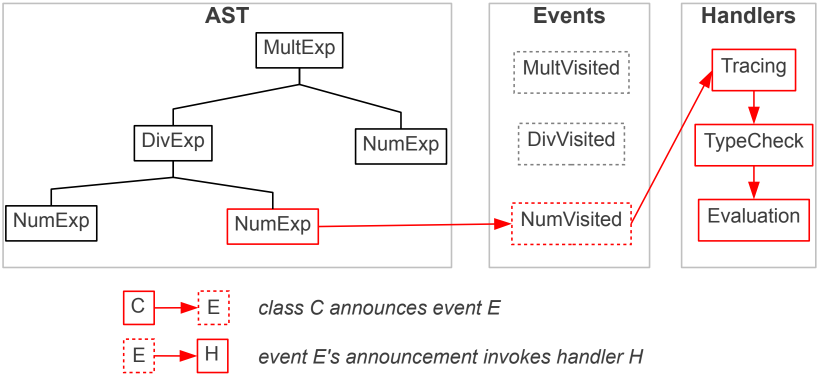


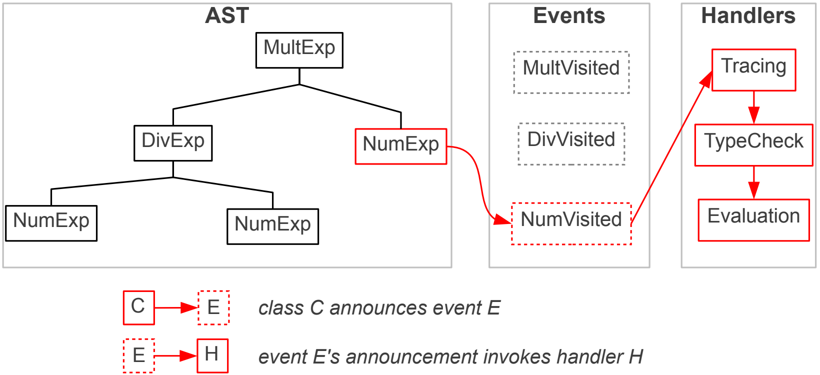
class C announces event E


AST
MultExp
DivExp
NumExp
NumExp
NumExp
Events
MultVisited
DivVisited
C
E
class C announces event E


AST
MultExp
DivExp
NumExp
NumExp
NumExp
Events
MultVisited
DivVisited
NumVisited
C
E
class C announces event E


AST
MultExp
DivExp
NumExp
NumExp
NumExp
Events
MultVisited
DivVisited
NumVisited
Handlers
Tracing
TypeCheck
Evaluation
C
E
class C announces event E
E
H
event E's announcement invokes handler H


AST
MultExp
DivExp
NumExp
NumExp
NumExp
Events
MultVisited
DivVisited
NumVisited
Handlers
Tracing
TypeCheck
Evaluation
C
E
class C announces event E
E
H
event E's announcement invokes handler H


AST
MultExp
DivExp
NumExp
NumExp
NumExp
Events
MultVisited
DivVisited
NumVisited
Handlers
Tracing
TypeCheck
Evaluation
C
E
class C announces event E
E
H
event E's announcement invokes handler H

MultVisited
<event>

+ node : MultExp
+ left : Exp
+ right : Exp

DivVisited
<event>

+ node : DivExp
+ left : Exp
+ right : Exp

PlusVisited
<event>

+ node : PlusExp
+ left : Exp
+ right : Exp

MultVisited
<event>

+ node : MultExp
+ left : Exp
+ right : Exp

DivVisited
<event>

+ node : DivExp
+ left : Exp
+ right : Exp

PlusVisited
<event>

+ node : PlusExp
+ left : Exp
+ right : Exp

MultVisited <event>
+ node : MultExp
+ left : Exp
+ right : Exp

DivVisited <event>
+ node : DivExp
+ left : Exp
+ right : Exp

PlusVisited <event>
+ 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;
}
```

```
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?

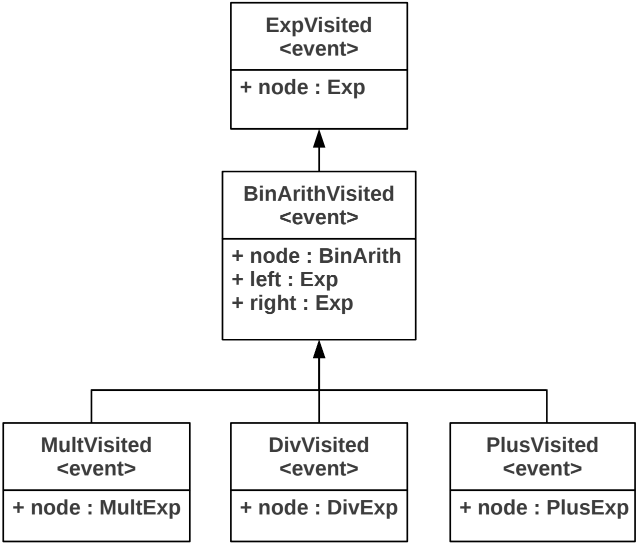
- Can we re-use code here?
  - **No!** Passing event closures (`next`) as argument is illegal. (to simplify reasoning about `invoke`/proceed functionality)
- What happens if a new AST type is added?
  - Must update **all** handlers to support that node type!
- What happens if an AST type is removed?

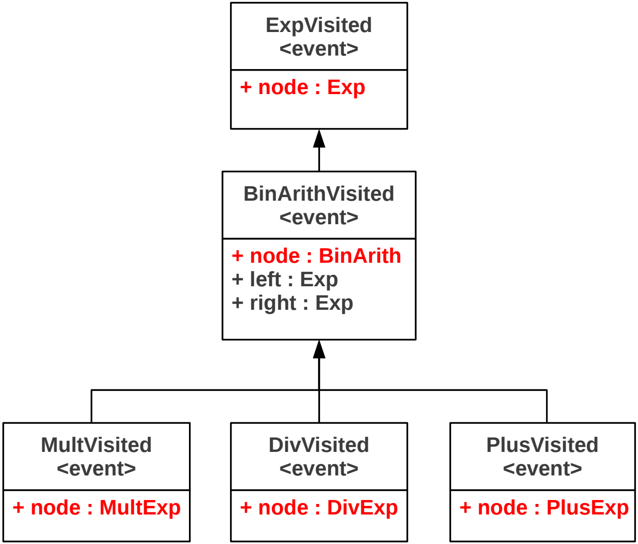
- Can we re-use code here?
  - **No!** Passing event closures (`next`) as argument is illegal. (to simplify reasoning about `invoke`/proceed functionality)
- What happens if a new AST type is added?
  - Must update **all** handlers to support that node type!
- What happens if an AST type is removed?
  - Must update **all** handlers and remove that node type!

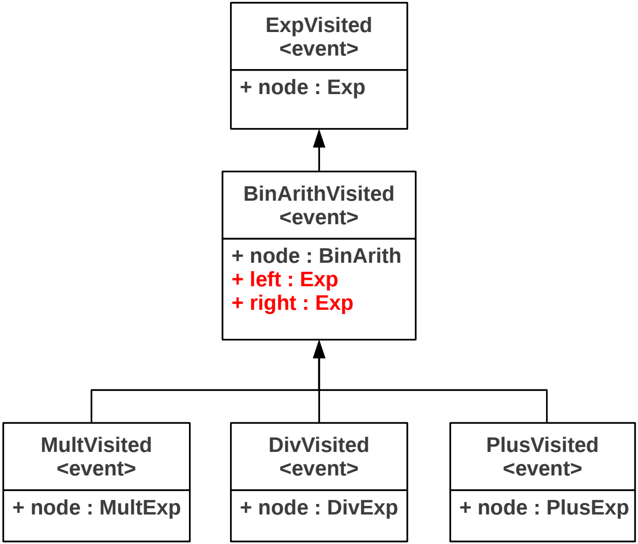
Ptolemy

- 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!


ExpVisited
<event>
+ node : Exp
BinArithVisited
<event>
+ node : BinArith
+ left : Exp
+ right : Exp
MultVisited
<event>
+ node : MultExp
DivVisited
<event>
+ node : DivExp
PlusVisited
<event>
+ node : PlusExp


ExpVisited
<event>
+ node : Exp
BinArithVisited
<event>
+ node : BinArith
+ left : Exp
+ right : Exp
MultVisited
<event>
+ node : MultExp
DivVisited
<event>
+ node : DivExp
PlusVisited
<event>
+ node : PlusExp


ExpVisited
<event>
+ node : Exp
BinArithVisited
<event>
+ node : BinArith
+ left : Exp
+ right : Exp
MultVisited
<event>
+ node : MultExp
DivVisited
<event>
+ node : DivExp
PlusVisited
<event>
+ node : PlusExp

```
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!

Ptolemy

```
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...

Ptolemy

$$
\begin{aligned}
\mathit{decl} ::= \;& \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \mathit{binding}^*\ \} \\
\mid\;& \boxed{c\ \texttt{event}\ p\ \boxed{\texttt{extends}\ q}\ \{\ \mathit{form}^*\ \}}
\end{aligned}
$$

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

$$binding ::= \texttt{when}\ p\ \texttt{do}\ m$$

$$\begin{aligned} e ::= &\ \texttt{register}(e) \mid \texttt{unregister}(e) \\ &\mid \texttt{announce}\ p\ (e^*)\ \{\ e\ \} \\ &\mid e\texttt{.invoke}() \end{aligned}$$

where

$m \in \mathcal{M}$, a set of method names

Ptolemy

(CHECK EVENT)

$$\frac{isClass(c) \qquad \forall i \in [1..n] :: isClass(t_i) \qquad \boxed{\mathsf{p} \ll: \mathsf{q}}}{\Pi \vdash c\ \mathtt{event}\ p\ \mathtt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\} : \mathsf{OK}}$$

Ptolemy

$$(\ll:\ \textsc{Top})\quad \frac{\mathit{isEvent}(p)}{p \ \ll:\ \ \mathit{Event}}$$

$$(\ll:\ \textsc{Reflexive})\quad \frac{\mathit{isEvent}(p)}{p \ \ll:\ \ p}$$

$$(\ll:\ \textsc{Transitive})\quad \frac{\mathit{isEvent}(q) \qquad \mathit{isEvent}(q') \qquad \begin{array}{c}\mathit{isEvent}(p) \\ p \ \ll:\ \ q'\end{array} \qquad q' \ \ll:\ \ q}{p \ \ll:\ \ q}$$

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

contextsOf recursively computes the list of all context for an event type q, 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/

Ptolemy

$$prog ::= decl^{*}\ e$$

$$decl ::= \texttt{class}\ c\ \texttt{extends}\ d\ \{\ field^{*}\ meth^{*}\ binding^{*}\ \}$$
$$\mid\ \boxed{c\ \texttt{event}\ p\ \boxed{\texttt{extends}\ q}\ \{\ form^{*}\ \}}$$

where

$$\begin{array}{lll} c & \in & \mathcal{C},\ \text{a set of class names} \\ d & \in & \mathcal{C} \cup \{Object\},\ \text{a set of superclass names} \\ & & \\ p & \in & \mathcal{P},\ \text{a set of event type names} \\ q & \in & \mathcal{P} \cup \{Event\},\ \text{a set of super event type names} \end{array}$$

$t ::= c \mid \boxed{\texttt{thunk}\ p}$

$field ::= c\ f$

$meth ::= c\ m\ (form^*)\ \{\ e\ \}$

$form ::= t\ var,$ where $var \neq \texttt{this}$

$\boxed{binding ::= \texttt{when}\ p\ \texttt{do}\ m}$

where

$f \in \mathcal{F}$, a set of field names

$m \in \mathcal{M}$, a set of method names

$var \in \{\texttt{this}\} \cup \mathcal{V}, \mathcal{V}$ is a set of variable names

$$
\begin{aligned}
ep ::= {} & n \mid var \mid ep.f \mid ep \mathrel{!=} \texttt{null} \mid ep == ep \\
& \mid ep < ep \mid \,!\, ep \mid ep \mathbin{\&\&} ep
\end{aligned}
$$

$$
\begin{aligned}
e ::= {} & \texttt{new}\ c() \mid var \mid \texttt{null} \mid e.m(e^*) \mid e.f \\
& \mid e.f = e \mid \texttt{cast}\ c\ e \mid form = e\ ;\ e \mid e\ ;\ e \\
& \mid \texttt{if}\ (ep)\ \{\ e\ \}\ \texttt{else}\ \{\ e\ \} \mid \texttt{while}\ (ep)\ \{\ e\ \} \\
& \mid \boxed{\texttt{register}(e) \mid \texttt{unregister}(e)} \\
& \mid \boxed{\texttt{announce}\ p\ (e^*)\ \{\ e\ \}} \\
& \mid \boxed{e.\texttt{invoke}()}
\end{aligned}
$$

where

$n \in \mathbb{Z}$, the set of integers

(CONCRETE TYPE INH.)

$$\frac{var'_i \notin \{var_1, \ldots, var_n\}}{concreteType(t'_i\ var'_i, [t_1\ var_1, \ldots, t_n\ var_n]) = t'_i\ var'_i}$$

(CONCRETE TYPE DEPTH)

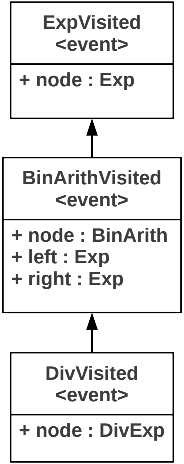
$$\frac{\exists j \in [1..n] :: t_j\ var'_i \in [t_1\ var_1, \ldots, t_n\ var_n]}{concreteType(t'_i\ var'_i, [t_1\ var_1, \ldots, t_n\ var_n]) = t_j\ var'_i}$$

(Top Context Vars)

$$\frac{}{contextsOf(Event) = \bullet}$$

(Context Vars)

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



contextsOf(ExpVisited) = [node:Exp]

contextsOf(BinArithVisited) = [node:BinArith, left:Exp, right:Exp]

contextsOf(DivVisited) = [node:DivExp, left:Exp, right:Exp]

$$\frac{(c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT}{isEvent(p)} \quad (\textsc{Is Event})$$

$$(\ll:\ \textsc{Top})\quad \frac{\mathit{isEvent}(p)}{p \ll: \mathit{Event}}$$

$$(\ll:\ \textsc{Refl.})\quad \frac{\mathit{isEvent}(p)}{p \ll: p}$$

$$(\ll:\ \textsc{Trans.})\quad \frac{\mathit{isEvent}(q) \qquad \mathit{isEvent}(q') \qquad \begin{array}{c}\mathit{isEvent}(p)\\ p \ll: q'\end{array} \qquad q' \ll: q}{p \ll: q}$$

$$(\ll:\ \textsc{Base})$$

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

Ptolemy

$\theta ::=$	"type attributes"
$\quad \text{OK}$	"program/top-level declaration"
$\quad \mid \text{OK in } c$	"method, binding"
$\quad \mid \texttt{var}\ t$	"var/formal/field"
$\quad \mid \texttt{exp}\ t$	"expression"
$\tau ::= c \mid \top \mid \bot$	"class type expressions"
$\pi, \Pi ::= \{I : \theta_I\}_{I \in K}$,	"type environments"

where K is finite, $K \subseteq (\mathcal{L} \cup \{\texttt{this}\} \cup \mathcal{V})$

(CHECK EVENT)

$$\frac{isClass(c) \qquad \forall i \in [1..n] :: isClass(t_i) \qquad \boxed{\mathrm{p} \lll: \mathrm{q}}}{\Pi \vdash c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, ..., t_n\ var_n\} : \mathrm{OK}}$$

(CHECK BINDING)

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

(ANNOUNCE EXP TYPE)

$$\frac{\begin{array}{c}(c\ \texttt{event}\ p\ \texttt{extends}\ q\ \{t_1\ var_1, \ldots, t_n\ var_n\}) \in CT \\ \forall i \in [1..n] :: \Pi \vdash e_i : \texttt{exp}\ t_i \qquad \Pi \vdash e : \texttt{exp}\ c' \qquad c' <: c\end{array}}{\Pi \vdash \texttt{announce}\ p(e_1, \ldots, e_n)\ \{e\} : \texttt{exp}\ c}$$

Ptolemy

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?