A Formalization of the Ptolemy II Type System

Chris Shaver, Marten Lohstroh

University of California, Berkeley
Typing in Ptolemy II
Ports Are Typed
Type Inference
Type Compatibility

Type compatibility is defined by a subtyping relation for connected ports. If a port of type $T_2$ is directly connected downstream to a port of type $T_1$:

$$T_1 \leq T_2$$

For instance, here $\text{int} \leq \text{double}$ meaning the output port of the Ramp is type compatible with the divide port of the MultiplyDivide actor.
The Type Lattice
Challenges with Typing
Example 1: Parametric Polymorphism

Intuitively, this would be parametric:

```
RecordAssembler

x : X, y : Y, output : {x : X, y : Y}
```

And variable instances of $X$ and $Y$ would simply be unified. But, instead it must be done with type inequalities:

```
x : X, y : Y, output : Z
{x : X, y : Y} \leq Z
Z.x \leq X
Z.y \leq Y
```
Example 2: Output Polymorphism

Bounded polymorphism could be used for outputs:

\[
\text{string} : \text{String}, \text{parse} : X \in \text{Parseable} \Rightarrow X
\]

Instances of parse could be written for each Parseable type, and type inference could determine the right instance from \(X\). This is currently done internally as part of the Actor implementation (runtime type-casing).
Backwards Type Inference
Example 3: Interfaces

We cannot define interface types. The token classes in Java have a different subclass hierarchy than their corresponding types in the type lattice i.e., $\tau_1 \leq \tau_2 \not\Rightarrow I_1 \subset I_2$. This causes problems when actors (like AddSubtract) bypass the Ptolemy II type system and instead resort to Java interfaces instead; automatic type conversion gets in the way.
Example 4: Higher-Order Types

There is currently no way to type check higher-order Actors:

Intuitively, one would like to type the Actor like this:

\[
\text{input : } X, \text{ model : } \text{Actor}\{\text{input : } X, \text{ output : } Y\}, \text{ output : } Y
\]
A Formalization
In order to describe typing formally, Ptolemy II models must be given a formal syntax.

### Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Symbol</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations</td>
<td>( r )</td>
<td>( { n_1, p_1, \ldots, n_k, p_k } ) where ( n_i ) are unique and ( p_i ) are ports.</td>
<td></td>
</tr>
<tr>
<td>Topologies</td>
<td>( t )</td>
<td>( \langle r_1, \ldots, r_k \rangle )</td>
<td></td>
</tr>
<tr>
<td>Clusters</td>
<td>( c )</td>
<td>( { n_1 \mapsto a_1, \ldots, n_k \mapsto a_k } ) where ( n_i ) are unique</td>
<td></td>
</tr>
<tr>
<td>Graphs</td>
<td>( g )</td>
<td>( c \parallel t )</td>
<td></td>
</tr>
<tr>
<td>Models</td>
<td>( m )</td>
<td>( d \Box g ) where ( d ) are Directors</td>
<td></td>
</tr>
<tr>
<td>Actors</td>
<td>( a )</td>
<td>( m</td>
<td>a_0 ) where ( a_0 ) are primitive Actors</td>
</tr>
</tbody>
</table>
Actor & Graph Types

In order to reason about Actor composition with formal types, an Actor type is created that is expressed in terms of the type of the attached ports. When an Actor is put in an Actor graph, it is labeled by a name $n$ and its ports $p$ are referenced by prefixing them with $n.p$:

$$
\begin{align*}
A & : \text{Actor}\{p_1 : \theta_1, \ldots, p_k : \theta_k\} \\
n & : \text{Graph}\{n_1.p_1 : \theta_1, \ldots, n_k.p_k : \theta_k\}
\end{align*}
$$

Whole models can be given the type of Actors, abstracting away the inner port types of the Actor Graph. This is consistent with the idea of Directors composing interfaces in Modular Actor Interfaces.
Constraint Quantification

Actor and graph types can be polymorphic, having ports assigned to type variables. These variables are quantified over constraints.

\[ \forall \bar{V}. C \Rightarrow \text{Actor}\{p_1 : \theta_1, \ldots, p_k : \theta_k\}, \]

where \( \theta_i \) is a type or type variable. For instance,

\[ \forall V_1, V_2, V_3. V_1 \leq V_2, V_1 \leq V_3 \Rightarrow \text{Actor}\{\text{control : boolean, input : } V_1, \text{ trueOutput : } V_2, \text{ falseOutput : } V_3\} \]

Constraints are given by the programmer or derived (if it is a composite).
Typing Rules

In typing rules the term syntax is connected to types by a set of typing judgements

\[ \Gamma \vdash t : T \]

Type derivations define these judgments, giving a conclusion rewritten from a set of premises:

\[ \Gamma_1 \vdash t_1 : T_1 \quad \Gamma_2 \vdash t_2 : T_2 \quad \ldots \quad \Gamma_N \vdash t_N : T_N \]

\[ \Gamma_N \vdash t_N : T_N \]
Full Formalization

Constraint Intro

\[ \forall \bar{V} . (C[\bar{V}] \vdash a : \tau[\bar{V}]) \]
\[ \vdash a : \forall \bar{V} . C \Rightarrow \tau \]

Graph Intro

\[ \forall k \leq K . (\vdash a_k : \forall \bar{V} . C_k \Rightarrow \text{Actor } S_k) \]
\[ \vdash \{n_1 \mapsto a_1, \ldots, n_k \mapsto a_k\} : \forall \bar{V} . \bigcup_{k \leq K} C_k \Rightarrow \text{Graph } \bigcup_{k \leq K} n_k . S_k \]

Topology Application

\[ \vdash c : \forall \bar{V} . C \Rightarrow \gamma \quad \forall a, b . R(a, b, t) \land P(a, b) \Rightarrow (\models \gamma. a \leq \gamma. b) \]
\[ \vdash c / t : \forall \bar{V} . C \cup \{\gamma. a \leq \gamma. b \mid \forall a, b . R(a, b, t) \Rightarrow Q(a, b)\} \Rightarrow \gamma \]

Director Application

\[ \vdash g : \forall \bar{V} . C \Rightarrow \text{Graph}\{@.p_1 : \theta_1, \ldots, @.p_k : \theta_k, \ldots\} \]
\[ \vdash d \boxdot g : \forall \bar{V} . C \Rightarrow \text{Actor}\{p_1 : \theta_1, \ldots, p_k : \theta_k\} \]

where

\[ R(a, b, t) \equiv \exists r \in t . (a, b) \in r \]
\[ P(a, b) \equiv \text{up}(a) \land \text{down}(b) \land \gamma. a \notin V_T \land \gamma. b \notin V_T \]
\[ Q(a, b) \equiv \text{up}(a) \land \text{down}(b) \land (\gamma. a \in V_T \lor \gamma. b \in V_T) \]
Example

Consider the following example:

Actors can be given the following types:

**Scale**: $\forall V_1, V_2. V_1 \leq V_2, V_2 \leq V_1 \leq \text{Scalar} \Rightarrow \text{Actor}\{input : V_1, output : V_2\}$

**Const**: $\text{Actor}\{output : \text{Int}\}$

**Asmb**: $\forall V_1, V_2, V_3. p \leq V_1, V_3. q \leq V_2, V_3 \leq \{p : V_1, q : V_2\} \Rightarrow \text{Actor}\{p : V_1, q : V_2, output : V_3\}$
Example

Consider the following example:

Using the typing rules the model can be given the following type:

\[
\d \uplus \{ P \mapsto \text{Scale}, \ Q \mapsto \text{Const}, \ R \mapsto \text{Asmb}\}
\]

\[
\langle \{ \uplus \text{.} x, \ P.\text{input} \}, \ \{ \ P.\text{output}, \ R.\text{p} \}, \ \{ \ Q.\text{output}, \ R.\text{q} \}, \ \{ \ R.\text{output}, \ @.y \} \rangle : \\
\forall V_{1..6} : V_{1} \leq V_{2}, \ V_{2} \leq V_{1}, \ V_{1} \leq \text{Scalar}, \ \{ p : V_{4}, \ q : V_{3} \} \leq V_{6}, \ V_{6}.p \leq V_{4}, \\
V_{6}.q \leq V_{5}, \ V_{7} \leq V_{1}, \ V_{2} \leq V_{4}, \ V_{3} \leq V_{5}, \ V_{6} \leq V_{8} \Rightarrow \text{Actor}\{ x : V_{7}, \ y : V_{8} \}
\]
Relation to other Type Systems

After working out the formal details, Christos Stergiou recognized the form of the type system as being similar to \(\text{HM}(X)\): Hindley-Milner over a constraint system. [Odersky 1999].

This is a success, since one of the main goals of this formalization was to draw comparisons to other formal type systems, and the wager was that someone would see it in this form and recognize it as a formal type system.
Potential Changes
Potential Changes and Future Work

This initial step of formalizing the type system opens the door for researching Ptolemy II typing in the context of formal type theory:

- Introduce ability to put `Actor` and `Graph` types into Base type constructors
  - This will allow strongly typed higher-order actors
- Add explicit unification; equality over type variables
- Introduce typeclass-based polymorphism
- Introduce dependent typing or existential quantification
- Formalize the constraints entirely
- Investigate the status of type checking and inference
- Perform reductions on constraint sets to type check polymorphic modules
- Investigate the semantics of automatic type conversion
Thanks!