Generic Process Shape Types and the POLY★ System

Jan Jakubův

ULTRA Group, MACS
Heriot-Watt University, Scotland

Seminar D3S, March 1, 2011, MFF, Prague
### Outline

**Process Calculi**
- Introduction
- The $\pi$-calculus
- Mobile Ambients

**Type Systems**
- Introduction
- Communication Safety in the $\pi$-calculus
- Communication Safety in MA

**Shape Types**

**Name Restriction**
- What is Name Restriction?
- Name Restriction with Shape Types

**Expressiveness**
- Direct Type Embeddings
- Alternative Expressiveness Evaluation
Outline

Process Calculi
- Introduction
- The $\pi$-calculus
- Mobile Ambients

Type Systems
- Introduction
- Communication Safety in the $\pi$-calculus
- Communication Safety in MA

Shape Types

Name Restriction
- What is Name Restriction?
- Name Restriction with Shape Types

Expressiveness
- Direct Type Embeddings
- Alternative Expressiveness Evaluation
What are Process Calculi?

- **concurrent systems** are environments where several interacting units engage in activity at the same time.
- **process calculi** are (one of) formal models of concurrent systems.
- **what are formal models for?**
  - to reason about systems
  - to verify their properties
  - to study the concepts of communication and interaction
Basics of Process Calculi

• models concurrency using rewriting systems
• provides syntax to represent units called processes \((P,Q)\)
• processes are used describe states of a system
• semantics of processes is given by a binary rewriting relation on processes \((P \rightarrow Q)\)
Example: the $\pi$-calculus

- introduced by Milner, Parrow, and Walker in 1992
- interaction is abstracted as a communication over named channels
- names ($a, b, \ldots$) represent both objects of communication and channel names
- two kinds of atomic actions ($A$):
  - $c \langle a \rangle$: send name $a$ over channel $c$
  - $c \langle x \rangle$: receive a name over channel $c$ and store it as $x$
\(\pi\)-processes

- processes are build from actions suing
  - sequential composition: \(A . P\)
  - parallel composition: \(P \parallel Q\)
- only one rewriting rule axiom:
  \[ c(x) . P \parallel c\langle a\rangle . Q \rightarrow P\{x \leftrightarrow a\} \parallel Q \]
Variant: polyadic $\pi$-calculus

- communicates tuples of names
- syntax: $c< a, b >$ and $c ( x, y, z )$
- communication errors can appear

$$c ( x, y, z ) . 0 \mid c < a, b > . 0$$

- communication safe process = without communication errors
Example: Mobile Ambients

- introduced by Cardelli and Gordon in 1998
- processes are placed in bounded locations, ambients \( (a[P]) \)
- an ambient contains processes and other ambients
- ambients form a tree hierarchy: \( a[b[P] | c[Q]] \)
Mobile Ambients Capabilities

- processes can change the ambient hierarchy by execution of capabilities
- in $a$: move into the sibling ambient $a$
- out $a$: exits the parent ambient $a$
- open $a$: dissolve the boundary of a child ambient $a$

$$a[\text{in } c.P_a] | d[\text{in } c.P_c] | \text{open } d.|0 \rightarrow$$
$$a[\text{in } c.P_a] | c[\text{in } c.P_c] \rightarrow$$
$$c[\text{in } c.P_c] | a[\text{in } c.P_a]$$
Mobile Ambients Communication

- processes in the same ambient can communicate (over anonymous channels)
- they can send names or capability sequences
  - \((x)\).open \(x.0\) \(\leftarrow\) \(<a>.0\) → open \(a.0\)
  - \((x)\).\(x.0\) \(\leftarrow\) \(<\text{in a.out b}>\) → in \(a.\text{out b}.0\)
MA Communication Errors

1. execution of a name: $a.0$

2. a name instead of a capability:

   $$(x).x.0 \mid <a>.0 \rightarrow a.0$$

3. a capability instead of a name:

   $$(x).in\ x.0 \mid <out\ a>.0 \rightarrow in\ (out\ a).0$$

- communication safe process = without communication errors
<table>
<thead>
<tr>
<th>Process Calculi</th>
<th>Type Systems</th>
<th>Shape Types</th>
<th>Name Restriction</th>
<th>Expressiveness</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The $\pi$-calculus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mobile Ambients</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Outline**

**Process Calculi**
- Introduction
- The $\pi$-calculus
- Mobile Ambients

**Type Systems**
- Introduction
- Communication Safety in the $\pi$-calculus
- Communication Safety in MA

**Shape Types**

**Name Restriction**
- What is Name Restriction?
- Name Restriction with Shape Types

**Expressiveness**
- Direct Type Embeddings
- Alternative Expressiveness Evaluation
What are Type Systems for Process Calculi?

- are used to statically verify various properties of processes (e.g. communication safety)
- usually one type system is handcrafted to verify a specific property
- properties have to be preserved by the rewriting relation $P \rightarrow Q$ (subject reduction)
- ... then it is enough to check the property for the initial state
Communication Safety in the $\pi$-calculus

- assign types $\tau$ to names
  1. atomic types ("Int", "String")
  2. channel types ("$\uparrow [\tau]$")
- context ($\Delta$) is a finite mapping from names to name types
- relation $\Delta \vdash P$ says that channels in $P$ are used consistently as described in $\Delta$
  - consider $\Delta = \{a \mapsto \text{Int}, b \mapsto \text{String}, c \mapsto \uparrow \text{[Int]}\}$
  - then "c<a>" is fine but "c<b>" is a communication error
Typing Rules

\[ \begin{align*}
\Delta \vdash 0 & \quad \Delta \vdash P \quad \Delta \vdash Q \\
\Delta \vdash P \mid Q & \quad \Delta \vdash \text{c}<\text{a}>.P \\
\Delta(\text{c}) = \uparrow [\Delta(\text{a})] & \quad \Delta \vdash P \\
\Delta \vdash \text{c}(\text{x}) . P & \quad \Delta \vdash \text{c}(\text{a} \mapsto \tau) . P \\
\end{align*} \]

- Theorem (Subject reduction):
  \[ \Delta \vdash P \quad \text{and} \quad P \rightarrow Q \quad \text{implies} \quad \Delta \vdash Q \]
Checking Communication Safety

- $\exists \Delta : \Delta \vdash P$ then $P$ is communication safe
- **over-approximation**: some communication safe processes are not recognized as safe
- $c<a>.0$ is safe and $\Delta = \{a \mapsto \alpha, c \mapsto \uparrow [\alpha]\}$
- $c<c>.0$ is safe but $\not\exists \Delta$
- $c(x, y).x<y>.0$
Communication Safety in MA

- assign a **single communication topic** to every ambient
- message types ($\omega$) = types of capability sequences:
  - $\text{Amb}[\kappa]$
  - $\text{Cap}[\kappa]$
- exchange type ($\kappa$) = types of processes:
  - $\text{Shh}$
  - $\omega_0 \times \cdots \times \omega_k$
Typing Relation

- **context** \((\Delta)\) = a finite function from ambient names to exchange types
- **typing relation** \(\Delta \vdash P : \kappa\)
- \(\exists (\Delta, \kappa) : \Delta \vdash P : \kappa\) then \(P\) is communication safe
- **over-approximation**: messenger ambient \((d, m) \cdot p [\text{in } d. <m>].0\)
Outline

Process Calculi
  Introduction
  The $\pi$-calculus
  Mobile Ambients

Type Systems
  Introduction
  Communication Safety in the $\pi$-calculus
  Communication Safety in MA

Shape Types

Name Restriction
  What is Name Restriction?
  Name Restriction with Shape Types

Expressiveness
  Direct Type Embeddings
  Alternative Expressiveness Evaluation
The POLY★ System

- introduced by Makholm and Wells in 2005
- POLY★ is a generic type system
- usually a type system is designed to verify a specific property in a specific process calculus
- POLY★ works for various process calculi
- POLY★ can verify various properties
- POLY★ is build on a generic notion of shape types
Instantiations of POLY★

- POLY★ needs to be instantiated to make a type system for a specific calculus
- this is done by describing rewriting rules in a special syntax
- the π-calculus communication rule

\[ c(x).P \ | \ c<a>.Q \rightarrow P\{x \leftrightarrow a\} \ | \ Q \]

is described as

`rewrite { \hat{c}(\hat{x}).\hat{P} \ | \ \hat{c}<\hat{a}>.\hat{Q} \leftarrow \{\hat{x} := \hat{a}\} \hat{P} \ | \ \hat{Q} }`
Shape Predicates

- are rooted graphs similar to syntax trees

\[ c<a>.0 \mid c(x).x<c>.0 \]
Shape Types are “closed” Shape Predicates

- all computational futures “smashed” together in one place

\[ \text{c}<\text{a}>.0 \mid \text{c}(x).x<\text{c}>.0 \]
Summary of Shape Types and POLY

- Shape types are graphs: can contain loops.
- POLY provides the algorithm to recognize types.
- POLY provides the implemented type inference algorithm.
- Advantages of Shape Types and POLY:
  - polymorphism
  - principal typings
  - type inference algorithm
# Outline

## Process Calculi
- Introduction
  - The $\pi$-calculus
  - Mobile Ambients

## Type Systems
- Introduction
  - Communication Safety in the $\pi$-calculus
  - Communication Safety in MA

## Shape Types

## Name Restriction
- What is Name Restriction?
- Name Restriction with Shape Types

## Expressiveness
- Direct Type Embeddings
- Alternative Expressiveness Evaluation
What is Name Restriction?

- name restriction makes some names private to a selected part of a process
- for example in the $\pi$-calculus, we can make some channel accessible only by a given process
- syntax "$\nu x. P$": $x$ is private in $P$ (scope)
- any "$x$" outside $P$ is different from $x$ in $P$:

$$(\nu c. c<a>.0) \mid c(a).0$$
\( \alpha \)-renaming of Private Names

- the scope of a private name can be extended (to allow interaction)
  \[
  (\nu x. P) \mid Q = \nu x. (P \mid Q)
  \]
- the scope can be extended only when \( x \) is not in \( Q \)
- when there is some \( x \) in \( Q \) then all occurrences of \( x \) in \( (\nu x. P) \) are renamed to a fresh name
- this is called \( \alpha \)-renaming:
  \[
  (\nu c. c^a.0) \mid c^a.0 = \nu d. (d^a.0 \mid c^a.0)
  \]
The Problem with $\alpha$-renaming

- when a private name is $\alpha$-renamed in a process then it escapes the corresponding name in the type

$$\nu c.c\langle a\rangle.0 = \nu d.d\langle a\rangle.0$$
Idea: Scope in Shape Types?

- it is not clear how to introduce scope to graphs

\[ \nu c. (c<a>.0 \mid c<b>.0) \neq (\nu c. c<a>.0) \mid c<b>.0 \]
Solution with Basic Names

- a name is a pair “a\textsubscript{i}” of basic name “a” and number “i”
- basic names are preserved under $\alpha$-renaming
- types are build only from basic names:

\[
(\nu c^0 . c^0 <a^0>.0) | d^0 <a^0>.0 \neq (\nu d^0 . d^0 <a^0>.0) | d^0 <a^0>.0
\]
<table>
<thead>
<tr>
<th>Process Calculi</th>
<th>Type Systems</th>
<th>Shape Types</th>
<th>Name Restriction</th>
<th>Expressiveness</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The $\pi$-calculus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mobile Ambients</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication Safety in the $\pi$-calculus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication Safety in MA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shape Types</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name Restriction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is Name Restriction ?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name Restriction with Shape Types</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Expressiveness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Type Embeddings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alternative Expressiveness Evaluation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing Type Systems - Type Embeddings

- Can we use generic POLY* shape types instead of types ($\tau$) of other systems?
- Can we embed type checking ($\vdash P : \tau$) in POLY* ($\vdash_*$) ?
- Direct type embedding: for a type $\tau$ construct the shape type $\tau^*$ such that

$$\vdash P : \tau \iff \vdash_* P : \tau^*$$
The Problem with Bound Names

- Direct type embedding is possible only when types of both type systems are “similar” enough.
- POLY shape types differ from types of many systems found in the literature.
- ...for example, in handling of bound names
- types of other systems do not contain “information” about bound names (their count, types, ...)

\[ \vdash \nu x. P : \tau \quad \iff \quad \vdash \nu y. P\{x \mapsto y\} : \tau \]
Greater Expressiveness of Shape Types

- Type systems are usually constructed to check a specific property of processes.
- In many cases this property can be expressed as a condition on shape types.
- Then we can use shape types to check the property directly and we do not need type embeddings.
- We use this approach to check communication safety in the $\pi$-calculus and Mobile Ambients:
  - $\text{POLY}^\star$ accepts all processes accepted by the other systems
  - $\text{POLY}^\star$ accepts some processes rejected by the other systems
Approach 1: Extended Embedding

- But, sometimes we want an **exact** embedding.
- We equip the type embedding $\tau^*$ with the required information about bound names (their number and types).
- These required information ($\inf_P$) are extracted from the process $P$ and thus the type embedding depends on $P$.
- We use this approach to encode basic Mobile Ambients types using shape types:

\[
\vdash P : \tau \iff \vdash^\star P : (\tau, \inf_P)^*
\]
Approach 2: Make Use of Principal Types

- Every process $P$ has a principal shape type $\Pi_P$.
- For every type of other system $\tau$ we define the property $P_\tau$ which exactly embeds the typing relation.
- We use this approach to encode basic channel types (Milner’s sorts) of the $\pi$-calculus:

$$\vdash P : \tau \iff P_\tau(\Pi_P)$$
Summary and Results

- We have extended POLY cryptocurrency to handle name restriction.
- We have evaluated the expressiveness of POLY cryptocurrency.
- Generic POLY cryptocurrency shape types
  - are more expressive than types of two selected systems handcrafted to verify a specific property
  - (but also) can express exactly the same properties as types of the two selected systems
POLY★ shape types web demonstration
http://macs.hw.ac.uk/ultra/polystar

Questions? Thank you...