ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Type Systems for Distributed Programs: Components and Sessions

by **Ornela Dardha**
PhD Supervisor: **Davide Sangiorgi**

May 19, 2014

Gentle Intro:
Features of Distributed Systems

- Complex Software Systems, in particular Distributed Systems, are everywhere around us.

Gentle Intro: Features of Distributed Systems

- Complex Software Systems, in particular Distributed Systems, are everywhere around us.
- Systems are highly mobile and dynamic: programs or devices move and new devices or pieces of software are added.

Gentle Intro: Features of Distributed Systems

- Complex Software Systems, in particular Distributed Systems, are everywhere around us.
- Systems are highly mobile and dynamic: programs or devices move and new devices or pieces of software are added.
- Systems are heterogeneous and open: pieces using different infrastructures and only partial knowledge of the system.

Gentle Intro: Features of Distributed Systems

- Complex Software Systems, in particular Distributed Systems, are everywhere around us.
- Systems are highly mobile and dynamic: programs or devices move and new devices or pieces of software are added.
- Systems are heterogeneous and open: pieces using different infrastructures and only partial knowledge of the system.
- Systems are designed as *structured composition of computational units* called components.

Gentle Intro:
Features of Distributed Systems

- Complex Software Systems, in particular Distributed Systems, are everywhere around us.
- Systems are highly mobile and dynamic: programs or devices move and new devices or pieces of software are added.
- Systems are heterogeneous and open: pieces using different infrastructures and only partial knowledge of the system.
- Systems are designed as *structured composition of computational units* called components.
- Giving rise to Component-Based Ubiquitous Systems (CBUS)

Distributed Systems in Practice

When reasoning about complex distributed systems, *reliability* and *usability* are of paramount importance.

Distributed Systems in Practice

When reasoning about complex distributed systems, *reliability* and *usability* are of paramount importance.

1. Reliability: Systems need to account for safe dynamic reconfiguration, namely changing at runtime the communication patterns.

Distributed Systems in Practice

When reasoning about complex distributed systems, *reliability* and *usability* are of paramount importance.

1. Reliability: Systems need to account for safe dynamic reconfiguration, namely changing at runtime the communication patterns.
2. Usability: Components perform communication among each-other, following predefined *patterns* or *protocols*.

Problem Description

We are interested in guaranteeing consistency and safety of CBUS.

Problem Description

We are interested in guaranteeing consistency and safety of CBUS.

1. Guaranteeing consistency of dynamic reconfigurations is a challenging task. It is difficult to ensure that modifications will not disrupt ongoing communications.

Problem Description

We are interested in guaranteeing consistency and safety of CBUS.

1. Guaranteeing consistency of dynamic reconfigurations is a challenging task. It is difficult to ensure that modifications will not disrupt ongoing communications.
2. Guaranteeing safety of communications means a collection of several requirements.
    - *privacy*
    - *communication safety*
    - *deadlock-freedom*
    - *livelock-freedom*

Aim of the Ph.D. Dissertation

To develop powerful techniques based on formal methods for the verification of correctness, consistency and safety properties related to dynamic reconfigurations and communications in complex distributed systems.

Approach

Static analysis based on Types and Type Systems. Why?

Approach

Static analysis based on Types and Type Systems. Why?

1. Types and Type Systems for safety properties.
    - concurrent programming: types for processes in the π-calculus
    - guarantee deadlock-freedom, livelock-freedom.

Approach

Static analysis based on Types and Type Systems. Why?

1. Types and Type Systems for safety properties.
   - concurrent programming: types for processes in the π-calculus
   - guarantee deadlock-freedom, livelock-freedom.

2. Types and Type Systems for communication.
   - ranging from *standard channel types* to *behavioural types*, like session types.
   - guarantee privacy, communication safety, session fidelity.

Contribution of the Ph.D. Dissertation

i) We design a type system for a component-based calculus, to *statically* ensure consistency of dynamic reconfigurations.

ii) We define an encoding of the π-calculus with session types into the standard typed π-calculus, to understand the expressive power of session types.

iii) We relate the notions of deadlock-freedom, livelock-freedom, progress defined in different calculi via the encoding.

Importance of the Contribution

i) Type System for Components:

1. Guarantees safe dynamic reconfiguration.
2. Shifts checks from *runtime* to *compile time*.

ii) Encoding of Session π-calculus:

1. Reusability of existing theory of the standard typed π-calculus.
2. Robustness by subtyping, polymorphism, HO and recursion.
3. Expressivity result for session types: not many results on types.

iii) Progress by Encoding:

1. Gives a systematic way of understanding the notions of deadlock-freedom, livelock-freedom, progress.
2. Encoding relates notions defined in different calculi.
3. Gives a new technique for guaranteeing progress.
4. More accurate analysis for progress property.

Publications

i) *Session Types Revisited*. O. Dardha, E. Giachino and D. Sangiorgi. In Proc. of PPDP'12, pp 139–150, ACM, 2012.

ii) *A Type System for Components*. O. Dardha, E. Giachino and M. Lienhardt. In Proc. of SEFM'13, pp 167–181, Springer LNCS, 2013.

iii) *Progress as Compositional Lock-Freedom*. M. Carbone, O. Dardha and F. Montesi. To appear in COORDINATION'14, Springer LNCS, 2014.

In the remainder...

- Safe Dynamic Reconfiguration
- Safe Communication by Encoding
- Progress of Communication

Safe Dynamic Reconfiguration

Component-Based Calculus

- Asynchronous Object Communication
  - Asynchronous method calls: $x = \mathrm{o}!\mathrm{m}(\mathit{args})$
  - Primitives to test and fetch the returned value.
- Concurrent Object Groups - cog
  - Cooperating Objects sharing the processor; only one task active at time.
  - A group's activity consists of a set of tasks, created by asynchronous method calls on objects of the group;
  - `new cog C()` creates a new object in a new group.
- Dynamic Reconfiguration
  - `rebind` $\mathrm{o.p} = \mathrm{o}'$ operation of `ports` of objects.

Component-Based Calculus in Practice: Clients, Server and Controller

```
...
Client c1 = new Client (s);
Client c2 = new cog Client (s);
Ctrl c = new Ctrl(c1,c2)!updateServer(s_new);
```

```
...
Unit updateServer(Server s_new) {
                                rebind c1.s = s_new;
                                rebind c2.s = s_new;
}
```

A Type System for Components

- Goal of the Type System
  1. check rebind performed internally to a cog.
  2. check synchronous method call performed internally to a cog.

- How do we do it?
  Statically track cogs identity and membership to a cog.

Component-Based Calculus in Practice: Clients, Server and Controller

```
...
Client c1 = new Client (s);← G
Client c2 = new cog Client (s); ← G′
Ctrl c = new Ctrl(c1,c2)!updateServer(snew);← G
```

```
...
Unit updateServer(Server snew) {
                                  rebind c1.s = snew;
                                  rebind c2.s = snew;
}
```

Component-Based Calculus in Practice: Clients, Server and Controller

```
...
Client c1 = new Client (s);← G
Client c2 = new cog Client (s); ← G'
Ctrl c = new Ctrl(c1,c2)!updateServer(snew);← G
```

```
...
Unit updateServer(Server snew) {
                              rebind c1.s = snew;   ok
                              rebind c2.s = snew;
}
```

Component-Based Calculus in Practice: Clients, Server and Controller

```
...
Client c1 = new Client (s);← G
Client c2 = new cog Client (s); ← G'
Ctrl c = new Ctrl(c1,c2)!updateServer(s_new);← G
```

```
...
Unit updateServer(Server s_new) {
                              rebind c1.s = s_new;   ok
                              rebind c2.s = s_new;   x
}
```

Properties of the Type System for Safe Dynamic Reconfiguration

Theorem (Main Result)

Well-typed programs do not perform

i) *illegal rebinding*

ii) *illegal synchronous method call*

Safe Communication by Encoding

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\text{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\text{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\boldsymbol{\nu} xy)\ \big(\texttt{server} \mid \texttt{client}\big)$$

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\texttt{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\texttt{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\boldsymbol{\nu} xy)\ \left(\texttt{server} \mid \texttt{client}\right)$$

Where

$$x : ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

and

$$y : !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\text{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\text{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\boldsymbol{\nu} xy)\ \left(\texttt{server} \mid \texttt{client}\right)$$

Where

$$x : ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

and

$$y : !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\texttt{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\texttt{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\nu xy)\ \big(\texttt{server} \mid \texttt{client}\big)$$

Where

$$x : ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

and

$$y : !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\text{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\text{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\boldsymbol{\nu} xy)\ \left(\texttt{server} \mid \texttt{client}\right)$$

Where

$$x : \texttt{?Int.?Int.!Bool.end}$$

and

$$y : \texttt{!Int.!Int.?Bool.end}$$

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\text{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\text{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\boldsymbol{\nu} xy)\ \left(\texttt{server} \mid \texttt{client}\right)$$

Where

$$x : \texttt{?Int.?Int.!Bool.end}$$

and

$$y : \texttt{!Int.!Int.?Bool.end}$$

Session Types in Practice: Equality Test

$$\texttt{server} \stackrel{\text{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\texttt{client} \stackrel{\text{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

The system is given by

$$(\boldsymbol{\nu} xy)\ \big(\texttt{server} \mid \texttt{client}\big)$$

Where

$$x : ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

and

$$y : !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

Session Types vs. Standard π- Types

- Session types are structured x : ?Int.?Int.!Bool.end;
- Standard π- channel types specify the type of the carried value: $x : \ell_i[\texttt{Int}]$ or $x : \ell_o[\texttt{Int}]$.
- Encoding is based on:
  1. Linearity of π- calculus channel types;
  2. Input/Output channel capabilities;
  3. Continuation-Passing principle.

Encoding Session Types

Let

$$S = ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

Then

$$[\![S]\!] = \ell_i[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Encoding Session Types

Let

$$S = ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

Then

$$[\![S]\!] = \ell_i[\mathtt{Int}, \ell_i[\mathtt{Int}, \ell_o[\mathtt{Bool}, \emptyset[]]]]$$

Encoding Session Types

Let

$$S = ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

Then

$$[\![S]\!] = \ell_i[\mathtt{Int}, \ell_i[\mathtt{Int}, \ell_o[\mathtt{Bool}, \emptyset[]]]]$$

Encoding Session Types

Let

$$S = \texttt{?Int.?Int.!Bool.end}$$

Then

$$[\![S]\!] = \ell_i[\texttt{Int}, \ell_i[\texttt{Int}, \ell_o[\texttt{Bool}, \emptyset[]]]]$$

Encoding Session Types

Let

$$S = ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

Then

$$[\![S]\!] = \ell_i[\mathtt{Int}, \ell_i[\mathtt{Int}, \ell_o[\mathtt{Bool}, \emptyset[]]]]$$

Properties of the Encoding

Theorem (On types)

Encoding preserves typability of programs.

Properties of the Encoding

Theorem (On types)

Encoding preserves typability of programs.

Theorem (On reductions)

Encoding preserves evaluation of programs.

Advanced Features on Safety by Encoding

Does the encoding handle extensions? Extend the calculi with:

- Subtyping
- Polymorphism
- Higher-Order
- Recursion

Advanced Features on Safety by Encoding

Does the encoding handle extensions? Extend the calculi with:

- Subtyping
- Polymorphism
- Higher-Order
- Recursion

Theorems 'On types' and 'On reductions' still hold.

Progress of Communication

Comparing Properties of Communication

- Deadlock-Freedom: communications eventually succeed, *unless the whole process diverges*. (Standard π)
- Livelock-Freedom: communications eventually succeed even if the whole process diverges. (Standard π)
- Progress: each *session*, once started, is guaranteed to satisfy all the requested interactions. (Session π)

What can we say about Progress?

Theorem

Progress is a compositional form of livelock-freedom property.

- We use the encoding to relate progress in the session π-calculus to livelock-freedom in the standard π- calculus.
- Reusability of type system and tools for livelock-freedom.
- More accurate analysis of the progress property.

Progress in Practice: “Bad” Process

Consider

$$(\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)(\ a?(z).d!\langle z\rangle \ \mid\ c?(w).b!\langle w\rangle\ )$$

Progress in Practice: “Bad” Process

Consider

$$(\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)(\ a?(z).d!\langle z\rangle \ \mid \ c?(w).b!\langle w\rangle\ )$$

By encoding we obtain the process:

$$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(\ x?(z).y!\langle z\rangle \ \mid \ y?(w).x!\langle w\rangle\ )$$

The type system for livelock-freedom rejects it!

Progress in Practice: “Good” Process

Consider the process

$$(\nu ab)\Big(\ b!\langle 1\rangle\ \ |\ \ (\nu cd)\big(\ d!\langle 1\rangle\ \ |\ \ c?(y).a?(z)\ \big)\Big)$$

Progress in Practice: "Good" Process

Consider the process

$$(\boldsymbol{\nu} ab)\Big(\ b!\langle 1\rangle \ \mid\ (\boldsymbol{\nu} cd)(\ d!\langle 1\rangle \ \mid\ c?(y).a?(z)\ )\Big)$$

By the encoding we obtain the process:

$$(\boldsymbol{\nu} k)\Big(\ k!\langle 1\rangle \ \mid\ (\boldsymbol{\nu} t)(\ t!\langle 1\rangle \ \mid\ t?(y).k?(z)\ )\ \Big)$$

The type system for livelock-freedom accepts it!

Conclusions and Future Work 1/2

Conclusions and Future Work 1/2

- **Problem**: guaranteeing consistency and safety properties in distributed programs.

Conclusions and Future Work 1/2

- **Problem**: guaranteeing consistency and safety properties in distributed programs.
- **Approach**: types and type systems.

Conclusions and Future Work 1/2

- **Problem**: guaranteeing consistency and safety properties in distributed programs.
- **Approach**: types and type systems.

i) Type system for safe dynamic reconfiguration in a concurrent object-oriented language for distributed systems.

Conclusions and Future Work 1/2

- **Problem**: guaranteeing consistency and safety properties in distributed programs.
- **Approach**: types and type systems.

i) Type system for safe dynamic reconfiguration in a concurrent object-oriented language for distributed systems.

ii) Encoding of session π-calculus into standard typed π-calculus permitting large reusability of existing theory and properties.

Conclusions and Future Work 1/2

- **Problem**: guaranteeing consistency and safety properties in distributed programs.
- **Approach**: types and type systems.

i) Type system for safe dynamic reconfiguration in a concurrent object-oriented language for distributed systems.

ii) Encoding of session π- calculus into standard typed π- calculus permitting large reusability of existing theory and properties.

iii) Progress in session π- calculus as livelock-freedom in standard typed π- calculus via encoding.

Conclusions and Future Work 2/2

- Type System for Components relevant in practice: designed for component-extension of ABS used in HATS and Envisage.
- Encoding of Session Types relevant for BETTY and ABCD.
- Extend the encoding to more general settings than dyadic session types, in particular multiparty session types.
- Session Types in Practice (ABCD)
- Tool for progress property in session types. Progress in more general settings.

Thank You!!

References I

During my PhD I produced the following 4 papers. The last one [4] is not part of my PhD thesis, as it resulted from my Master's work.

Marco Carbone, Ornela Dardha, and Fabrizio Montesi.
Progress as compositional lock-freedom.
In *COORDINATION*, volume 8459 of *LNCS*, pages 49–64. Springer, 2014.

Ornela Dardha, Elena Giachino, and Michael Lienhardt.
A Type System for Components.
In *Software Engineering and Formal Methods - 11th International Conference, SEFM*, volume 8137 of *LNCS*, pages 167–181. Springer, 2013.

References II

Ornela Dardha, Elena Giachino, and Davide Sangiorgi.
Session types revisited.
In *PPDP*, pages 139–150, New York, NY, USA, 2012. ACM.

Ornela Dardha, Daniele Gorla, and Daniele Varacca.
Semantic Subtyping for Objects and Classes.
In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec*, volume 7892 of *LNCS*, pages 66–82. Springer, 2013.

Component Extension of Core ABS 1/2

$$
\begin{array}{rcl}
P & ::= & \overline{Dl}\ \{\ s\ \} \\
Dl & ::= & D\ \mid\ F\ \mid\ I\ \mid\ C \\
T & ::= & \mathtt{V}\ \mid\ \mathtt{D}[\langle \overline{T} \rangle]\ \mid\ (\mathtt{I}, r) \\
r & ::= & \bot\ \mid\ \mathtt{G}[\overline{f : T}]\ \mid\ \alpha\ \mid\ \mu\alpha.r \\
D & ::= & \textbf{data}\ \mathtt{D}[\langle \overline{T} \rangle] = \mathtt{Co}[(\overline{T})]\overline{|\mathtt{Co}[(\overline{T})]}; \\
F & ::= & \textbf{def}\ T\ \mathtt{fun}[\langle \overline{T} \rangle](\overline{T\ x}) = e; \\
I & ::= & \textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x;}\ \overline{S}\ \} \\
C & ::= & \textbf{class}\ \mathtt{C}[(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\ \overline{M}\ \} \\
Fl & ::= & [\textbf{port}]\ T\ x \\
S & ::= & [\textbf{critical}]\ (\mathcal{G}, r)\ T\ \mathtt{m}(\overline{T\ x}) \\
M & ::= & S\ \{\ s\ \}
\end{array}
$$

Component Extension of Core ABS 1/2

$$
\begin{array}{lcl}
s & ::= & \textbf{skip} \mid s\ ;\ s \mid T\ x \mid x = z \mid \textbf{await}\ g \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \{\ s\ \} \mid \textbf{return}\ e \\
 & \mid & \textbf{rebind}\ e.p = z \mid \textbf{suspend} \\
z & ::= & e \mid \textbf{new}\ [\textbf{cog}]\ \mathtt{C}\ (\overline{e}) \mid e.\mathtt{m}(\overline{e}) \mid e!\mathtt{m}(\overline{e}) \mid \textbf{get}(e) \\
e & ::= & v \mid x \mid \mathtt{fun}(\overline{e}) \mid \textbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} \mid \mathtt{Co}[(\overline{e})] \\
v & ::= & \mathtt{true} \mid \mathtt{false} \mid \textbf{null} \mid \mathtt{Co}[(\overline{v})] \\
p & ::= & \_ \mid x \mid \textbf{null} \mid \mathtt{Co}[(\overline{p})] \\
g & ::= & e \mid e? \mid \|e\| \mid g \wedge g
\end{array}
$$

Standard π- types

$\tau ::=$	$\emptyset[\widetilde{T}]$	channel with no capability
	$\ell_i[\widetilde{T}]$	linear input
	$\ell_o[\widetilde{T}]$	linear output
	$\ell_\sharp[\widetilde{T}]$	linear connection
$T ::=$	τ	linear channel type
	$\langle l_i\_T_i \rangle_{i \in I}$	variant type
	$\sharp T$	standard channel type
	$\texttt{Bool}$	boolean type
	$\cdots$	other constructs

Session Types

$q ::=$	$\texttt{lin} \mid \texttt{un}$	qualifiers
$p ::=$	$!T.U$	send
	$?T.U$	receive
	$\oplus\{l_i : T_i\}_{i \in I}$	select
	$\&\{l_i : T_i\}_{i \in I}$	branch
$T ::=$	$q\ p$	qualified pretype
	$\texttt{end}$	termination
	$\texttt{Bool}$	boolean type

Encoding of session types

$$
\begin{aligned}
[\![\mathtt{end}]\!] &\stackrel{\mathtt{def}}{=} \emptyset\,[] && \text{(E-End)}\\
[\![!T.U]\!] &\stackrel{\mathtt{def}}{=} \ell_o[[\![T]\!], [\![\overline{U}]\!]] && \text{(E-Out)}\\
[\![?T.U]\!] &\stackrel{\mathtt{def}}{=} \ell_i[[\![T]\!], [\![U]\!]] && \text{(E-Inp)}\\
[\![\oplus\{l_i : T_i\}_{i\in I}]\!] &\stackrel{\mathtt{def}}{=} \ell_o[\langle l_i\_[\![\overline{T_i}]\!]\rangle_{i\in I}] && \text{(E-Select)}\\
[\![\&\{l_i : T_i\}_{i\in I}]\!] &\stackrel{\mathtt{def}}{=} \ell_i[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] && \text{(E-Branch)}
\end{aligned}
$$

Encoding of session processes

$$
\begin{aligned}
[\![\mathbf{0}]\!]_f &\stackrel{\text{def}}{=} \mathbf{0} \\
[\![x!\langle v\rangle.P]\!]_f &\stackrel{\text{def}}{=} (\boldsymbol{\nu} c)f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\
[\![x?(y).P]\!]_f &\stackrel{\text{def}}{=} f_x?(y, c).[\![P]\!]_{f,\{x\mapsto c\}} \\
[\![x \triangleleft l_j.P]\!]_f &\stackrel{\text{def}}{=} (\boldsymbol{\nu} c)f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\
[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f &\stackrel{\text{def}}{=} f_x?(y).\ \textbf{case}\ y\ \textbf{of}\ \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I} \\
[\![\textbf{if}\ v\ \textbf{then}\ P\ \textbf{else}\ Q]\!]_f &\stackrel{\text{def}}{=} \textbf{if}\ f_v\ \textbf{then}\ [\![P]\!]_f\ \textbf{else}\ [\![Q]\!]_f \\
[\![P \mid Q]\!]_f &\stackrel{\text{def}}{=} [\![P]\!]_f \mid [\![Q]\!]_f \\
[\![(\boldsymbol{\nu} xy)P]\!]_f &\stackrel{\text{def}}{=} (\boldsymbol{\nu} c)[\![P]\!]_{f,\{x,y\mapsto c\}}
\end{aligned}
$$

Subtyping in standard π- calculus

$$\frac{}{T \leq T}\,(\mathrm{S}\pi\text{- }\textsc{Refl}) \qquad \frac{T \leq T' \qquad T' \leq T''}{T \leq T''}\,(\mathrm{S}\pi\text{- }\textsc{Trans})$$

$$\frac{\widetilde{T} \leq \widetilde{T'}}{\ell_i[\widetilde{T}] \leq \ell_i[\widetilde{T'}]}\,(\mathrm{S}\pi\text{- }\mathtt{ii}) \qquad \frac{\widetilde{T'} \leq \widetilde{T}}{\ell_o[\widetilde{T}] \leq \ell_o[\widetilde{T'}]}\,(\mathrm{S}\pi\text{- }\mathtt{oo})$$

$$\frac{I \subseteq J \qquad T_i \leq T_j' \quad \forall i \in I}{\langle l_i\_T_i\rangle_{i\in I} \leq \langle l_j\_T_j'\rangle_{j\in J}}\,(\mathrm{S}\pi\text{- }\textsc{Variant})$$

Polymorphism

Example of polymorphism in the π- calculus with/without sessions:

$$x : !\langle X; D\rangle.\texttt{end}\ ,\ y : ?\langle X; D\rangle.end$$

$$\vdash x!\langle Int; 5\rangle \mid y?(z).\ \texttt{open}\ z\ \texttt{as}\ (X; w)\ \texttt{in}\ nj!\langle w\rangle$$

$$\longrightarrow\ \texttt{open}\ \langle Int; 5\rangle\ \texttt{as}\ (X; w)\ \texttt{in}\ nj!\langle w\rangle$$

$$\longrightarrow nj!\langle 5\rangle$$

Semantics of Bounded Polymorphism

$$(\boldsymbol{\nu} xy)(x \triangleleft l_j(B).P \mid y \triangleright \{l_i(X_i <: B_i) : P_i\}_{i \in I} \mid R) \rightarrow$$
$$(\boldsymbol{\nu} xy)(P \mid P_j[B/X_j] \mid R) \quad j \in I$$

$$\textbf{case}\, l_j(B)\_v\ \textbf{of}\ \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i \in I} \rightarrow P_j[B/X_j][v/x_j] \quad j \in I$$

Higher-order constructs

$\sigma ::=$	T	general type
	$\diamondsuit$	process type
$T ::=$	$\texttt{Unit}$	unit type
	$T \to \sigma$	functional type
	$T \xrightarrow{1} \sigma$	linear functional type
$P ::=$	PQ	application
	v	values
$v ::=$	$\lambda x : T.P$	abstraction
	$\star$	unit value

Encoding Higher-Order

$$\begin{aligned}
[\![T \xrightarrow{1} \sigma]\!] &\stackrel{\texttt{def}}{=} [\![T]\!] \xrightarrow{1} \sigma \\
[\![T \rightarrow \sigma]\!] &\stackrel{\texttt{def}}{=} [\![T]\!] \rightarrow \sigma
\end{aligned}$$

$$\begin{aligned}
[\![\lambda x : T.P]\!]_f &\stackrel{\texttt{def}}{=} \lambda x : [\![T]\!].[\![P]\!]_f \\
[\![PQ]\!]_f &\stackrel{\texttt{def}}{=} [\![P]\!]_f[\![Q]\!]_f
\end{aligned}$$

Where $\sigma ::= T \mid \Diamond$

On progress for sessions

Definition (Progress)

A process P has *progress* if for all $\mathcal{C}[\cdot]$ such that $\mathcal{C}[P]$ is well-typed, $\mathcal{C}[P] \to^* \mathcal{E}[R]$ (where R is an input or an output) implies that there exist $\mathcal{C}'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $\mathcal{C}'[\mathcal{E}[R]] \to^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu} xy)$ is a restriction in $\mathcal{C}'[\mathcal{E}[R]]$.

Results for Progress

Theorem (Progress $\Leftrightarrow$ Lock-freedom)

Let P be a well-typed closed process. Then P is livelock-free if and only if P has progress.

Theorem (Progress $\Leftrightarrow$ Closed Lock-Free)

If P is well-typed then P has progress if and only if $\texttt{close}(\texttt{P})$ *is livelock-free.*

Typing Progress

procedure $\textsc{Progress}(\Gamma, P)$
  Check $\Gamma \vdash P$
  Build $\texttt{close(P)}$ from Γ
  Encode $[\![\texttt{close(P)}]\!]_f = \texttt{P}'$
  return TyPiCal(P')
end procedure