Session-Types

Typing Your Channels Since 1993

Renato Costa    Nico Ritschel
November 28, 2018

University of British Columbia
1. Motivation
2. Type Systems
3. Session Types: Introduction
4. Linear Logic
5. Linear Session Types
6. Implementations
Motivation
Figure 1: Interacting with an ATM: depositing and withdrawing money
A Session Type Example: Interacting with an ATM

\[
ch : \text{Nat} \to \mu S . \bigoplus \\
\begin{align*}
\text{deposit} : \text{Nat} . S \\
\text{withdraw} : \text{Nat} . & \\
\text{dispense} : S \\
\text{overdraft} : S \\
\text{balance} : \text{Nat} . S \\
\text{quit} : \text{end}
\end{align*}
\]
Type Systems
Recap

\( n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \)

t ::= ... 

T ::= \text{Nat} \mid \text{Bool}

\[\begin{align*}
\vdash \text{true} : \text{Bool} & \quad \vdash \text{false} : \text{Bool} & \quad \vdash n : \text{Nat} \\
\vdash t_1 : \text{Bool} & \quad \vdash t_2 : T & \quad \vdash t_3 : T \\
& \quad \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{Bool} \\
\vdash t : \text{Nat} & \quad \vdash \text{pred}(t) : \text{Nat} & \quad \vdash \text{zero?}(t) : \text{Bool}
\end{align*}\]
Progress:
A well-typed program never gets stuck. It either is a value or it can step further.

Preservation:
If a well-typed program steps, the resulting program remains well-typed.

Soundness:
If a program is well-typed, it behaves during run-time according to its syntactic type.
Type Environments

\[ n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \]

\[ x \in \text{VAR} \quad \Gamma \in \text{TENV} = \text{VAR}^{\text{fin}} \rightarrow \text{TYPE} \]

\[ t ::= \ldots \]

\[ T ::= \text{Nat} \mid \text{Bool} \]

\[ \Gamma \vdash \text{true} : \text{Bool} \]
\[ \Gamma \vdash \text{false} : \text{Bool} \]
\[ \Gamma \vdash n : \text{Nat} \]

\[ \Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \]

\[ \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{Bool} \]

\[ \Gamma \vdash t : \text{Nat} \]

\[ \Gamma \vdash \text{succ}(t) : \text{Nat} \]

\[ \Gamma \vdash \text{pred}(t) : \text{Nat} \]

\[ \Gamma \vdash \text{zero?}(t) : \text{Bool} \]
Type Environments

\[ n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \]
\[ x \in \text{VAR} \quad \Gamma \in \text{TENV} = \text{VAR}^{\text{fin}} \rightarrow \text{TYPE} \]
\[ t ::= \ldots \mid \text{let} \; x = t \; \text{in} \; t \]
\[ T ::= \text{Nat} \mid \text{Bool} \]

\[ \Gamma \vdash \text{true} : \text{Bool} \quad \Gamma \vdash \text{false} : \text{Bool} \quad \Gamma \vdash n : \text{Nat} \]
\[ \Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \]
\[ \Gamma \vdash \text{if} \; t_1 \; \text{then} \; t_2 \; \text{else} \; t_3 : \text{Bool} \]
\[ \Gamma \vdash t : \text{Nat} \quad \Gamma \vdash \text{succ}(t) : \text{Nat} \quad \Gamma \vdash \text{pred}(t) : \text{Nat} \quad \Gamma \vdash \text{zero?}(t) : \text{Bool} \]

\[ \Gamma \vdash t_1 : T' \quad \Gamma[x \mapsto T'] \vdash t_2 : T \]
\[ \Gamma \vdash \text{let} \; x = t_1 \; \text{in} \; t_2 : T \]
\[ \Gamma \vdash x : \Gamma(x) \quad x \in \Gamma \]
The Curry-Howard Correspondence says that type systems are isomorphic to logical proofs. This means that the following structural rules implicitly hold:

Exchange:

\[ \Gamma, A, B \vdash C \]
\[ \Gamma, B, A \vdash C \]

Weakening:

\[ \Gamma \vdash C \]
\[ \Gamma, A \vdash C \]

Contraction:

\[ \Gamma, A, A \vdash C \]
\[ \Gamma, A \vdash C \]
Product Types

\[ n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \]

\[ x \in \text{VAR} \quad \Gamma \in \text{TENV} = \text{VAR}^{\text{fin}} \rightarrow \text{TYPE} \]

\[ t ::= \ldots \mid [t, t] \mid t[1] \mid t[2] \]

\[ T ::= \ldots \mid T \times T \]

\[ \Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \]

\[ \Gamma \vdash [t_1, t_2] : T_1 \times T_2 \]

\[ \Gamma \vdash t : T_1 \times T_2 \]

\[ \Gamma \vdash t[1] : T_1 \]

\[ \Gamma \vdash t[2] : T_2 \]
Type Ascription

\[ n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \]
\[ x \in \text{VAR} \quad \Gamma \in \text{TEnv} = \text{VAR} \xrightarrow{\text{fin}} \text{TYPE} \]
\[ t ::= ... \mid t \text{ as } T \]
\[ T ::= ... \]

\[ \vdash t : T \]
\[ \vdash t \text{ as } T : T \]
Sum Types

\[ \begin{align*}
\text{n} & \in \mathbb{N} \quad \text{t} \in \text{TERM} \quad \text{T} \in \text{TYPE} \\
\text{x} & \in \text{VAR} \quad \Gamma & \in \text{TENV} = \text{VAR} \xrightarrow{\text{fin}} \text{TYPE} \\
t & ::= \ldots \mid \text{in}_l \ t \mid \text{in}_r \ t \mid \text{case} \ t \ of \ x \mapsto t \ or \ x \mapsto t \\
\text{T} & ::= \ldots \mid \text{T} + \text{T}
\end{align*} \]

\[ \begin{array}{c}
\frac{\Gamma \vdash t : T_1}{\Gamma \vdash \text{in}_l \ t : T_1 + T_2} \quad \frac{\Gamma \vdash t : T_1}{\Gamma \vdash \text{in}_r \ t : T_1 + T_2}
\end{array} \]

\[ \begin{align*}
\Gamma & \vdash t_0 : T_1 + T_2 \\
\Gamma[x_1 \mapsto T_1] & \vdash t_1 : T \\
\Gamma[x_2 \mapsto T_2] & \vdash t_2 : T
\end{align*} \]

\[ \\
\Gamma \vdash \text{case} \ t_0 \ of \ x_1 \mapsto t_1 \ or \ x_2 \mapsto t_2 : T
\]

Example: let \( x = (\text{if} \ldots \text{then} \text{in}_1 \ 1 \ \text{else} \text{in}_2 \ \text{false}) \) in

\[
\text{case} \ x \ of \ x_1 \mapsto \text{succ}(x) \\
\quad \text{or} \ x_2 \mapsto 0
\]
Sum Types

\[ n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \]
\[ x \in \text{VAR} \quad \Gamma \in \text{TENV} = \text{VAR} \xrightarrow{\text{fin}} \text{TYPE} \]
\[ t ::= \ldots | \text{in}_l \ t \ | \text{in}_r \ t \ | \text{case} \ t \ \text{of} \ x \mapsto t \ \text{or} \ x \mapsto t \]
\[ T ::= \ldots | T + T \]

\[
\begin{align*}
\Gamma \vdash t : T_1 & \quad \Gamma \vdash t : T_1 \\
\Gamma \vdash \text{in}_l \ t : T_1 + T_2 & \quad \Gamma \vdash \text{in}_r \ t : T_1 + T_2 \\
\Gamma \vdash t_0 : T_1 + T_2 & \quad \Gamma [x_1 \mapsto T_1] \vdash t_1 : T \\
\Gamma [x_2 \mapsto T_2] \vdash t_2 : T \\
\Gamma \vdash \text{case} \ t_0 \ \text{of} \ x_1 \mapsto t_1 \ \text{or} \ x_2 \mapsto t_2 : T
\end{align*}
\]

Problem: Unbound \( T_1 \) and \( T_2 \) require "guessing" or leave typing underspecified!
Sum Types

\[ n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \]

\[ x \in \text{VAR} \quad \Gamma \in \text{TENV} = \text{VAR} \overset{\text{fin}}{\rightarrow} \text{TYPE} \]

\[ t ::= \ldots \mid \text{in}_l \ t \ as \ T \mid \text{in}_r \ t \ as \ T \mid \text{case} \ t \ of \ x \mapsto t \ or \ x \mapsto t \]

\[ T ::= \ldots \mid T + T \]

\[ \Gamma \vdash t : T_1 \]

\[ \Gamma \vdash \text{in}_l \ t \ as \ T_1 + T_2 : T_1 + T_2 \]

\[ \Gamma \vdash t : T_1 \]

\[ \Gamma \vdash \text{in}_r \ t \ as \ T_1 + T_2 : T_1 + T_2 \]

\[ \Gamma \vdash t_0 : T_1 + T_2 \quad \Gamma[x_1 \mapsto T_1] \vdash t_1 : T \quad \Gamma[x_2 \mapsto T_2] \vdash t_2 : T \]

\[ \Gamma \vdash \text{case} \ t_0 \ of \ x_1 \mapsto t_1 \ or \ x_2 \mapsto t_2 : T \]

Solution: Require type ascriptions to make other type option explicit.
Recursive Types

\[ n, t, T \in \text{TERM, TYPE} \]
\[ x, \Gamma \in \text{VAR, TENV} = \text{VAR} \xrightarrow{\text{fin}} \text{TYPE} \]
\[ t ::= ... \]
\[ T ::= ... \mid X \mid \mu X . T \]

Example: \( \mu \text{List} . (\text{Num} \times \text{List}) + \text{Num} \)

There are two ways to define recursive types:

- **Equi-Recursive**, which means that unfolding the recursion is done on demand (similar to using a fix-point combinator for Recursive functions).
- **Iso-recursive**, which means that points for unfolding the recursion are manually defined by the user.
(Iso-)Recursive Types

\( n \in \mathbb{N} \quad t \in \text{TERM} \quad T \in \text{TYPE} \)

\( x \in \text{VAR} \quad \Gamma \in \text{TENV} = \text{VAR} \xrightarrow{\text{fin}} \text{TYPE} \)

\( t ::= \ldots \mid \text{fold} [T] \ t \mid \text{unfold} [T] \ t \)

\( T ::= \ldots \mid X \mid \mu X.T \)

\[
\begin{align*}
\Gamma \vdash t : [X \Rightarrow U] \ T_1 & \quad U = \mu X.T \\
\Gamma \vdash \text{fold} [T] \ t_1 : U & \\
& \Gamma \vdash \text{unfold} [T] \ t_1 : [X \Rightarrow U] \ T_1 \quad U = \mu X.T
\end{align*}
\]

Example:

\( \text{unfold} [\mu \text{List} . (\text{Num} \times \text{List}) + \text{Num}] \ t \)

\( : \mu \text{List} . (\text{Num} \times \mu \text{List} . (\text{Num} \times \text{List}) + \text{Num}) + \text{Num} \)
We (informally) define abbreviations:

List = \( \mu L . \ (\text{Num} \times L) + \text{Num} \)

ListBody = (\text{Num} \times \text{List}) + \text{Num}

Example: let list = fold [List] in

\[
\begin{align*}
\text{fold [List] in}_l & \ [1, \\
& \quad \text{fold [List] in}_r \ 2 \text{ as ListBody}] \text{ as ListBody in} \\
& \quad \text{case unfold [List] list of} \\
& \quad \quad \text{cons}_1 \mapsto \\
& \quad \quad \quad \quad \text{(case unfold [List] cons}_1.2 \text{ of} \\
& \quad \quad \quad \quad \quad \text{cons}_2 \mapsto \text{cons}_2.1 \\
& \quad \quad \quad \quad \quad \text{or last} \mapsto \text{last) } \\
& \quad \quad \text{or last} \mapsto 0
\end{align*}
\]

What does this program evaluate to?
We (informally) define abbreviation:

\[ \text{List} = \mu \text{L}. (\text{Num} \times \text{L}) + \text{Num} \]

Example: let \( \text{list} = \text{in}_l [1, \text{in}_r 2 \text{ as List}] \text{ as List} \) in case list of

\[ \text{cons}_1 \mapsto \]

(case \( \text{cons}_1.2 \text{ of} \)

\[ \text{cons}_2 \mapsto \text{cons}_2.1 \]

or last \( \mapsto \text{last} \))

or last \( \mapsto 0 \)

What does this program evaluate to?
Algebraic Data Types (ADTs)

List = Cons Num List | Last Num

Example: let list = Cons [1, Cons [2, Last 3]] in
            match list
            case Cons first rest₁:
                 (match rest₁
                    case Cons second rest₂: second
                    case Last last: last)
            case Last last: 0

Languages like Haskell use ADTs to restrict recursive types to be able to reason about them, and to provide syntactic sugar for \( \text{in}_l \) and \( \text{in}_r \).
Session Types: Introduction
Session Types

- Created by Kohei Honda in 1993 [5]

- A type system for Milner’s $\pi$-Calculus

- Formalizes the interaction between processes that communicate by exchanging messages

- For simplicity, we will be considering binary sessions in the following slides (communication between exactly two processes)
Definition for a session-typed system.

\[ n \in \mathbb{N} \quad S \in \text{SESSIONTYPE} \quad T \in \text{TYPE} \quad X \in \text{TVAR} \quad l \in \text{LABEL} \]

\[ S ::= \ ?[T].S \mid ![T].S \mid \&\{ l : S, l : S, \ldots, l : S \} \]
\[ \mid \bigoplus\{ l : S, l : S, \ldots, l : S \} \mid \text{end} \]
\[ \mid X \mid \mu X.S \]

\[ T ::= \text{Nat} \mid \text{Bool} \]

Note: Labels are just for convenience reasons. If we use a fixed set of options (e.g. 2) for \( \bigoplus \) and \( \& \), we can leave them out.
Definition for a session-typed system.

\[ n \in \mathbb{N} \quad S \in \text{SESSIONTYPE} \quad T \in \text{TYPE} \quad X \in \text{TVAR} \quad l \in \text{LABEL} \]

\[
S ::= \ ?[T].S \ | \ ![T].S \ | \ \&\{ \ l : S , l : S , \ldots , l : S \ \} \\
| \ \bigoplus\{ \ l : S , l : S , \ldots , l : S \ \} \ | \ \text{end} \\
| \ X \ | \ \mu X . S
\]

\[
T ::= \ \text{Nat} \ | \ \text{Bool}
\]

- **Obs:** If we defined \( T \) to be \( T ::= S | \text{Nat} | \text{Bool} \), then we would be able to send channels along channels, a property called **mobility** in the \( \pi \)-calculus, or **higher-order channels** in the Session Types literature.
Session Types: Connectives

Session types are used to specify the communication protocol between two parties. The syntax for session types is defined as follows:

\[ n \in \mathbb{N} \quad S \in \text{SESSIONTYPE} \quad T \in \text{TYPE} \quad X \in \text{TVAR} \quad l \in \text{LABEL} \]

Session types are defined inductively:

\[ S ::= \ ?[T].S \quad | \quad ![T].S \quad | \quad &\{ \ l : S, \ l : S, \ldots, \ l : S \ \} \]
\[ \quad | \quad \oplus\{ \ l : S, \ l : S, \ldots, \ l : S \ \} \quad | \quad \text{end} \]

**Rule**

- \(?[T].S\):
  - **Meaning**: Receive input of type \(T\) and proceed as \(S'\)

- \( ![T].S \):
  - **Meaning**: Send output of type \(T\) and proceed as \(S'\)

- \&\{ \ l : S, \ l : S, \ldots, \ l : S \ \}:
  - **Meaning**: Choices given to the client

- \oplus\{ \ l : S, \ l : S, \ldots, \ l : S \ \}:
  - **Meaning**: Choices provided by the client

- \text{end}:
  - **Meaning**: Process terminates
Session Types: Connectives

\[ n \in \mathbb{N} \quad S \in \text{SESSIONTYPE} \quad T \in \text{TYPE} \quad X \in \text{TVAR} \quad l \in \text{LABEL} \]

\[ S ::= \ ?[T].S \quad \mid \quad ![T].S \quad \mid \quad \&\{ \ l : S , l : S , \ldots , l : S \} \]
\[ \mid \quad \oplus\{ \ l : S , l : S , \ldots , l : S \} \quad \mid \quad \text{end} \]

- The pairs \{!, ?\} and \{\&, \oplus\} are **duals**
- It helps to think of \& as **external choice** and \(\oplus\) as **internal choice** (from point of view of the provider)
Revisiting our example: Interacting with an ATM

User process uses a channel of type $\text{ch}_u$ to communicate with ATM.

$\text{ch}_u : \forall \text{Nat}. \mu S. \oplus$

\[
\begin{align*}
\text{deposit} : & \forall \text{Nat}. S \\
\text{withdraw} : & \forall \text{Nat}. \&
\end{align*}
\]

\[
\begin{align*}
\text{dispense} : & S \\
\text{overdraft} : & S \\
\text{balance} : & ?\forall \text{Nat}. S \\
\text{quit} : & \text{end}
\end{align*}
\]
What is the type of the channel the ATM process uses to communicate with User?
**Interacting with an ATM**

ATM process uses a channel $\text{ch}_a$ that is the dual of $\text{ch}_u$ to communicate with User.

$$\text{ch}_a : \text{?Nat} \cdot \mu S \cdot \& \begin{cases} \text{deposit} : \text{?Nat.S} \\ \text{withdraw} : \text{?Nat.} \oplus \{ \text{dispense} : S \\ \text{overdraft} : S \} \\ \text{balance} : !\text{Nat.S} \\ \text{quit} : \text{end} \end{cases}$$
A valid interaction between User and ATM

Let $\text{ch}_u$ be a channel as defined previously. Before any message is exchanged, it has the type as shown before.

The user then proceeds and sends its identifier:

$$\text{ch}_u ! 42$$

The type of $\text{ch}_u$ is now:

$$\text{ch}_u : \mu S . \Theta \left\{ \begin{array}{l} \text{deposit} : !\text{Nat}.S \\
\text{withdraw} : !\text{Nat} . & \{ \begin{array}{l}
\text{dispense} : S \\
\text{overdraft} : S \\
\text{balance} : ?\text{Nat}.S \\
\text{quit} : \text{end} \end{array} \end{array} \right\}$$
Types of channels **change** as computation progresses and messages are exchanged.
After our user sends its identifier (and before she sends any other message), what happens if another user also wants to interact with our ATM and sends its own identifier?
In the context of Session Types, preservation means that if the types of the client and provider are duals initially and continue to be duals throughout the computation.

- **Expectations** of clients and providers are always aligned.

- This is commonly referred to as **session fidelity**.
Currently, we allow processes that communicate using our session-typed channels to form a graph.

For our purposes, we want to stop channels from being shared by multiple processes.

We need control over channel resources.
Linear Logic
• Created by Girard in 1987 [4]

• Linearity is an important aspect in concurrent systems and, in particular, behavioral type systems.

• Linear Logic is a substructural logic where propositions are treated as resources.
Weakening and Contraction

Since Linear Logic treats propositions as resources, weakening and contraction are **not** admissible.

**Weakening-left**

\[
\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}
\]

**Weakening-right**

\[
\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}
\]

**Contraction-left**

\[
\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}
\]

**Contraction-right**

\[
\frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}
\]
### Connectives of Linear Logic

<table>
<thead>
<tr>
<th>Connective</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow B$</td>
<td>Linear implication</td>
</tr>
<tr>
<td>$A \otimes B$</td>
<td>Multiplicative conjunction</td>
</tr>
<tr>
<td>$A &amp; B$</td>
<td>Additive conjunction</td>
</tr>
<tr>
<td>$A \oplus B$</td>
<td>Additive disjunction</td>
</tr>
</tbody>
</table>
Linear Session Types
Linear Logic and Session Types

- Introduced by Caires and Pfenning in 2010 [1].
  - Interpretation of binary session types in intuitionistic linear logic

- Correspondence between **linear logic propositions** and **session types**

- Communication discipline interpreted as reductions in **logical derivations**
Type Judgements

Type judgements are of the form:

$$\Gamma; \Delta \vdash P :: z : S$$

where:

- $\Gamma$ is the **non-linear** typing context (resources subject to weakening and contraction)
- $\Delta$ is the **linear** typing context
- $P$ is a **process**
- $z$ is a **channel**
- $S$ is a **session type**
Some Typing Rules of Linear Session Types

External Choice

\[
\Gamma; \Delta \vdash P :: z : A \quad \Gamma; \Delta \vdash Q :: z : B
\]
\[
\Gamma; \Delta \vdash [P, Q] :: z : A \land B
\]

\[
\Gamma; \Delta, x : A \vdash P :: z : C
\]
\[
\Gamma; \Delta, x : A \land B \vdash x[1].P :: z : C
\]

\[
\Gamma; \Delta, x : B \vdash P :: z : C
\]
\[
\Gamma; \Delta, x : A \land B \vdash x[2].P :: z : C
\]
Some Typing Rules of Linear Session Types

Internal Choice

\[
\frac{\Gamma; \Delta \vdash P :: z : A}{\Gamma; \Delta \vdash z.\text{in}_l. P :: z : A \oplus B}
\]

\[
\frac{\Gamma; \Delta \vdash P :: z : B}{\Gamma; \Delta \vdash z.\text{in}_r. P :: z : A \oplus B}
\]

\[
\frac{\Gamma; \Delta, x : A \vdash P :: z : C}{\Gamma; \Delta, x : A \oplus B \vdash \text{case } x \text{ of } P \text{ or } Q :: z : C}
\]

\[
\frac{\Gamma; \Delta, x : B \vdash Q :: z : C}{\Gamma; \Delta, x : A \oplus B \vdash \text{case } x \text{ of } P \text{ or } Q :: z : C}
\]
Implementations
Session Types: Implementations

- Most mainstream languages have *some* way of supporting session types
  - A more comprehensive list can be found in [2]

- Implementations are typically classified as:
  - **Primitive** vs. **Library** vs. **External**
  - **Static** vs. **Dynamic** vs. **Hybrid**
Case Study: Gong

- Session types implementation for **Go channels** [6, 7]

- Goal: statically detect (local and global) deadlocks, as well as liveness issues.

- **External** tool. Sessions types are **inferred** using static analysis. Model checker looks for **safety and liveness** violations.
package main

import {
    "fmt"
    "time"
}

func Work() {
    for {
        fmt.Println("Working")
        time.Sleep(1 * time.Second)
    }
}

func Send(ch chan<int>) { ch < 42 }
func Recv(ch < chan int, done chan<int>) { done < < ch }

func main() {
    ch, done := make(chan int), make(chan int)
    go Send(ch)
    go Recv(ch, done)
    go Recv(ch, done)
    go Work()
    < done
    < done
}

Example: Local Deadlock
Figure 2: Program has a liveness violation! (Good luck now finding the bug)


