COMS 6998 TLC Project Proposal: Session Types ("Session")

Anderi Coman, Christopher Yoon

{ac4808, cjy2129}@columbia.edu

1 Abstract

We are interested in building an interpreter for the π-calculus, with **session types** and **subtyping**. Specifically, we will be implementing the language specified in *Subtyping for session types in the pi calculus* from the work of *Simon Gay* and *Malcolm Hole* (2005) [1].

We demonstrate through our implementation of the language how the structure of communication between two sessions can be specified as a type, and their correctness be statically checked via a type system.

2 Brief Language Specification

We will describe the language with a simple example. Consider the following program:

```
(ν x: ^[str]) (x+!["ping"].x+?[s: str].end || x-?[s: str].x-!["pong"].end)
```

First, (ν x: ^[str]) adds a new string-typed bidirectional channel (named `x`) to the scope/environment. Then, the two processes `(x+!["ping"].x+?[s: str].end)` and `(x-?[s: str].x-!["pong"].end)` (call them P_1 and P_2) are executed in parallel. Parallelism is indicated by the `||` operator.

At any moment, the two polarities of the channel `x` (written as `x+` and `x-`) are said to be "owned" by exactly one process each - although this is not evident from the syntax, the typing rules enforce that one process cannot own both endpoints; this restriction only prohibits pathological programs which contain deadlocks. Concurrently, P_1 sends (marked by `!`) the string `"ping"` through `x`, and process P_2 waits for and receives it (marked by `?`). P_2 then sends `"pong"` through `x`, while P_1 waits for and receives it. After that, both processes terminate.

As mentioned in the beginning, channel `x` was a "regular" string-typed bidirectional channel. A better way to enforce this protocol would be via a session-typed channel:

```
(ν x: ![str].?[str]) (x+!["ping"].x+?[s: str].end) || (x-?[s: str].x-!["pong"].end)
```

We say that `x+` has the session type `![str].?[str].end` for "sends a string, receives a string, then terminates," and `x-` has the "dual" session type `?[str].![str].end`, for "receives a string, sends a string, then terminates." In particular, the two communication protocols are compatible (no processes become stuck), and the program is *correct*. It is worth noting that the types associated with names `x+` and `x-` change as the communication unfolds. Intuitively, each communication operation "consumes" the prefix of that respective endpoint's type so that, in the end, all channels should be completely "consumed" (i.e. should have type `end`).

This compatibility is enforced via the subtyping relationship. In general terms, it expresses what types of communication are acceptable through a channel of a given type. For instance, a server which offers a large set of choices should be able to communicate with a client which can only request a smaller set (given that the channel through which they communicate has an appropriate type). So, the type system associated with this version of the pi-calculus is, in some sense, more general since it allows for subtyping. The subtyping relationship defined in this paper can be checked algorithmically, which allows for efficient type checking.

If P_1 attempted to communicate with an incompatible process, as in the following examples

```
(ν x: ![str].?[str]) (x+!["ping"].x+?[s: str].end) || (x-!["pong"].x-?[s: str].end)

(ν x: ![str].?[str]) (x+!["ping"].x+?[s: str].end) || (x-?[s1: str].x-!["pong"].x-?[s2: str].end)

(ν x: ![str].?[str]) (x+!["ping"].x+?[s: str].end) || (x-?[s: str].x-![123].end)
```

then the session types themselves would be incompatible, leading to a typing error.

Lastly, we note that, although our examples here differentiate between the two polarities of each channel by using the superscripts +/-, this distinction need not be made explicitly; polarities can be inferred by the type checking system. This idea will be reflected in our final grammar.

More formally, the language can be described by the following syntax rules. In the interest of time and space, we omit the other specifications (operation semantics and typing rules) in the proposal.

Session Types	$S ::= X$	type variable
	$\mid$ end	terminated session
	$\mid ?[T_1, \ldots, T_n].S$	input
	$\mid ![T_1, \ldots, T_n].S$	output
	$\mid \&\langle \ell_1 : S_1, \ldots \ell_n : S_n \rangle$	branch
	$\mid \oplus \langle \ell_1 : S_1, \ldots \ell_n : S_n \rangle$	choice
	$\mid \mu X.T$	recursive session type

Types	$T ::= X$	type variable
	$\mid S$	session type
	$\mid \hat{\ }[T_1, \ldots, T_n]$	standard channel type
	$\mid \mu X.T$	recursive channel type

Figure 1: Syntax of Types

$P, Q ::= \mathbf{0}$	terminated process
$\mid P \mid\mid Q$	parallel combination
$\mid !P$	replication
$\mid x^p?[y_1 : T_1, \ldots, y_n : T_n].P$	input
$\mid x^p![y_1^{p_1}, \ldots, y_n^{p_n}].P$	output
$\mid (\nu x : T)P$	channel creation
$\mid x^p \triangleright \{\ell_1 : P_1, \ldots, \ell_n : P_n\}$	branch
$\mid x^p \triangleleft \ell.P$	choice

Figure 2: Syntax of Processes

3 Deliverables and Implementation

So far, we have implemented the communication-primitives for a simple π-calculus interpreter with the `OCaml` language. The concurrency and channel communication primitives of the π-calculus language are taken care of by our `OCaml` backend, using the `Async` library. We intend to either (a) use these primitives to simulate parallel evaluation concurrently or (b) reduce all programs simultaneously in a single thread (similar to the evaluation of the Lambda-calculus via beta-reductions). As for extending the π-calculus interpreter with session types and subtyping, we have implemented the front-end (lexing, parsing), as well as the type substitution and the duality/subtyping relations.

Our remaining steps are:

1. Complete the interpreter.
2. Implement session type checking.
3. Implement a more human-friendly programming language based on our π-calculus interpreter, whose specifications are not yet thought-out thoroughly. We are considering integrating features of the lambda-calculus.

References

[1] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3): 191–225, 2005. doi: 10.1007/s00236-005-0177-z. URL `https://doi.org/10.1007/s00236-005-0177-z`.