COMS 6998 TLC Project: Session Types (“Session”)

Anderi Coman, Christopher Yoon

{ac4808, cjy2129}@columbia.edu

1 Abstract

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

We implement the interpreter with the `OCaml` language, and demonstrate through it how communication protocols (sessions) can be specified formulated as a type system, allowing the correctness of inter-thread communication in multi-threaded programs be statically checked.

Our implementation can be found at https://github.com/session-pi/session-pi.

2 A Motivating Example

Before going into the formal specification of the language, we describe the language with simple example programs. Consider the following:

```
(ν x: ^[str]) (x+!["ping"].x+?[s: str].0 | x-?[s: str].x-!["pong"].0)
```

Figure 1: A motivating example

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].0)` and `(x-?[s: str].x-!["pong"].0)` (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 is known as linearity. The polarities of the channel can be inferred by the interpreter, and thus can be omitted. The linearity of channels only prohibits pathological programs which contain deadlocks.

In the program, P_1 sends (marked by `x![...]`) the string `"ping"` through `x`, and process P_2 waits for and receives it (marked by `x?[...]`). Upon receiving `"ping"`, P_2 then sends `"pong"` through `x`, while P_1 waits for and receives it. After that, both processes terminate, indicated by the keyword `0`.

As such, our language is primarily concerned with the communication of concurrent processes. The more important features of the language (such as session types and subtyping) will be discussed in more detail in the following sections.

3 Formal Specification of the Language

3.1 Syntax

The syntax of the language, specified in *Gay and Hole* [1], can be described formally as followed:

$P, Q ::= \mathbf{0}$	terminated process
$\mid\ P \mid Q$	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

Most of the components were already introduced in the motivating example 1; we now explain the additional elements.

First, replication $!P$ denotes forking a process P *when needed*. That is, replication happens lazily, when the context (other processes that are blocked) specifically demands it. $!P$ is congruent to $P \mid !P$ and is evaluated as such in the operational semantics. This is particularly useful when modeling server-client programs where the server has to execute the same communication protocols for an arbitrary number of clients:

```
!(x!["hello"].0) | x?[msg: String].0 | x?[msg: String].0
```

In the example above, the server process (`x!["hello"].0`) replicates itself twice to serve the two clients.

Next, branches allow for a process to execute multiple communication protocols; correspondingly, a `choice` process can choose among the protocols that the `branch` process offers. For instance, consider a "vending-machine server" `branch` process, which can serve a free-coke or a Dr. Pepper:

```
x |> {freecoke: v!["coke"].0, drpepper: v?[money: Int].v!["drpepper"].0 }
```

A customer (another process) can choose which protocol to execute:

```
x |> {freeCoke: v!["coke"].0, drpepper: v?[m: Int].v!["drpepper"].0 } |
  x <| freeCoke.v?[itm: String]
```

While the operational semantics will be discussed later, the above program will execute as followed:

```
v!["coke"].0 | v?[itm: String].0
0 | 0
0
```

where each line above indicates each step of the reduction.

3.2 The Type System

Our type system supports the following types:

Session Types	$S ::= X$	type variable
	$\mid\ \mathsf{end}$	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
	$\mid\ \mathit{Int} \ \mid\ \mathit{Bool}$	primitive type
	$\mid\ S$	session type
	$\mid\ \hat{}[T_1, \ldots, T_n]$	standard channel type
	$\mid\ \mu X.T$	recursive channel type

Figure 3: Syntax of Types

A standard channel c of type $\hat{}[\vec{T}]$ is an unlimited resource through which communicated data must always have type $\vec{T}$ (i.e. at every I/O operation, several values whose number and types correspond to $\vec{T}$ are either sent or received through the channel). A standard channel's endpoints are not differentiated, can be replicated and sent across multiple processes. A session-typed channel, on the other hand has exactly two associated endpoints with polarities $+$ and $-$, each of which must be owned by exactly one process. The types of the two endpoints are dual to each other, where the duality relationship is defined as follows:

$$\overline{X} = X \qquad \overline{\mathit{Int}} = \mathit{Int} \qquad \overline{\mathit{Bool}} = \mathit{Bool}$$

$$\begin{aligned}
\overline{\mathsf{end}} &= \mathsf{end} \\
\overline{?[\vec{T}].S} &= ![\vec{T}].\overline{S} \\
\overline{![\vec{T}].S} &= ?[\vec{T}].\overline{S} \\
\overline{\&\langle l_i : S_i \rangle_{i \in [n]}.S} &= \oplus \langle l_i : \overline{S_i} \rangle_{i \in [n]}.S \\
\overline{\oplus \langle l_i : S_i \rangle_{i \in [n]}.S} &= \& \langle l_i : \overline{S_i} \rangle_{i \in [n]}.S \\
\overline{\mu X.S} &= \mu X.\overline{S}
\end{aligned}$$

Figure 4: Dual Types

Their corresponding types are linear and indicate the sequence of operations which must be performed on a channel at every moment of time. Note that, although session-typed channels have associated polarities for each of their endpoints, these polarities can be omitted from the syntax of the processes and inferred at type-checking, while imposing minimal additional restrictions on expressiveness.

3.2.1 Linear Types

The type system implements session-types as linear (note, however, that standard channel types are not linear). More precisely, for every I/O operation performed on a session-channel name, the name is removed from the environment and re-added with an "advanced"/"stepped" type. Every such channel must be either consumed completely by the owning process or sent through a channel to a different process (in which case the ownership is passed to that respective process). Otherwise, the process is not typed correctly and the typechecking fails.

One particular restriction imposed by the linearity of session types concerns the 'Rep' operator (!). Since every channel endpoint must be owned by exactly one channel at any time and must eventually reach the 'end' type, no session-typed channel name is made available to a 'Rep' process, and every instance of a 'Rep' process must either concede ownership of the session-typed channels it creates or advance them to the 'end' type.

3.2.2 Recursive Types

The type syntax provides support for arbitrarily-defined recursive types. A type variable introduced in the type definition of a specific name extends only within the scope of that single type definition. We also note that type variables (which enable the creation of recursive types) can be defined either in the context of regular types or session types. As such, every type variable is only available within the same context it is created (i.e. a type variable denoting a regular type cannot be used as a session-typed, but can be used as the type passed through a session-typed channel).

One restriction imposed upon recursive types is that left-recursive types (e.g. $\mu T.\mu S.S$) are disallowed, as progress on their corresponding names would be impossible.

3.2.3 Subtyping

The subtyping relation of our type system is implemented according to the following rules:

$$\frac{}{\text{Int} \le \mathit{Int}}\ \text{S-INT} \qquad \frac{}{\text{Bool} \le \mathit{Bool}}\ \text{S-BOOL}$$

$$\frac{}{\text{end} \le \mathit{end}}\ \text{S-END} \qquad \frac{\forall i.\,(T_i \le U_i) \quad \forall i.\,(U_i \le T_i)}{\hat{}[\vec{T}] \le \hat{}[\vec{U}]}\ \text{S-CHAN}$$

$$\frac{\forall i.\,(T_i \le U_i) \quad V \le W}{?[\vec{T}].V \le ?[\vec{U}].W}\ \text{S-INS} \qquad \frac{\forall i.\,(U_i \le T_i) \quad V \le W}{![\vec{T}].V \le ![\vec{U}].W}\ \text{S-OUTS}$$

$$\frac{m \le n \quad \forall i.\,(S_i \le T_i)}{\&\langle l_i : S_i\rangle_{i\in[m]} \le \&\langle l_i : T_i\rangle_{i\in[n]}}\ \text{S-BRANCH}$$

$$\frac{m \le n \quad \forall i.\,(S_i \le T_i)}{\oplus\langle l_i : S_i\rangle_{i\in[n]} \le \oplus\langle l_i : T_i\rangle_{i\in[m]}}\ \text{S-CHOICE}$$

Figure 5: Subtyping Relationship of Non-recursive types

These types are based on the principle of safe substitution (i.e. if $S \le T$, then a value of type S can be inserted anywhere a value of type T is expected). The subtyping relationship requires that $?, \&$ are covariant, $!, \oplus!$ are contravariant, and regular channel types are invariant. The subtyping relation of recursive types involves the use of a context which memorizes all the subtyping relationships inferred thus far. Subtyping a recursive type starts with the assumption that the two type variables denoting said types are part of the subtyping relationship. The subtyping relationship is incorporated into the algorithmic typing rules as detailed below.

$$\frac{}{\Gamma \vdash_X \mathbf{0} : \{x^p \in X \mid \Gamma(x^p) = \text{end}\}}\ \text{TC-NIL}$$

$$\frac{\Gamma \vdash_X \boldsymbol{P} : Y \quad \Gamma - Y \vdash_{X-Y} \boldsymbol{Q} : Z}{\Gamma \vdash_X \boldsymbol{P} \mid \boldsymbol{Q} : Y \cup Z}\ \text{TC-PAR}$$

$$\frac{\Gamma \vdash_\emptyset \boldsymbol{P} : \emptyset}{\Gamma \vdash_X \boldsymbol{!P} : \emptyset}\ \text{TC-REP} \qquad \frac{\Gamma, x : T \vdash_X \boldsymbol{P} : Y}{\Gamma \vdash_X (\boldsymbol{\nu x : T})\boldsymbol{P} : Y}\ \text{TC-NEW}$$

$$\frac{\Gamma, x^+ : S, x^- : \bar{S} \vdash_{X \cup \{x^+, x^-\}} \boldsymbol{P} : Y \quad \{x^+, x^-\} \subseteq Y \quad \text{end} \not\leq S \in SType}{\Gamma \vdash_X (\boldsymbol{\nu x : S})\boldsymbol{P} : Y - \{x^+, x^-\}}\ \text{TC-NEWS}$$

$$\frac{\Gamma, x : {}^\frown[\vec{T}], \vec{y} : \vec{U} \vdash_{X \cup Y^S} \boldsymbol{P} : Y \quad Y^S \subseteq Y \quad \vec{T} \leq \vec{U}}{\Gamma, x : {}^\frown[\vec{T}] \vdash_X \boldsymbol{x?[\vec{y} : \vec{U}].P} : Y - Y^S}\ \text{TC-IN}$$

$$\frac{\left(\Gamma, x : {}^\frown[\vec{T}]\right) - Y^S \vdash_{X - Y^S} \boldsymbol{P} : Y \quad Y^S \subseteq X \quad \left(\Gamma, x : {}^\frown[\vec{T}]\right)[\vec{y}] \leq \vec{T}}{\Gamma, x : {}^\frown[\vec{T}] \vdash_X \boldsymbol{x![\vec{y}].P} : Y \cup Y^S}\ \text{TC-OUT}$$

$$\frac{\Gamma, x : S, \vec{y} : \vec{U} \vdash_{X \cup Y^S} \boldsymbol{P} : Y \quad Y^S \subseteq Y \quad \vec{T} \leq \vec{U} \quad x \in X \cap Y}{\Gamma, x : ?[\vec{T}].S \vdash_X \boldsymbol{x?[\vec{y} : \vec{U}].P} : Y - Y^S}\ \text{TC-INS1}$$

$$\frac{\Gamma, x^+ : S, \vec{y} : \vec{U} \vdash_{X \cup Y^S - \{x^-\}} \boldsymbol{P} : Y \quad Y^S \subseteq Y \quad \vec{T} \leq \vec{U} \quad x^+ \in X \cap Y}{\Gamma, x^+ : ?[\vec{T}].S \vdash_X \boldsymbol{x?[\vec{y} : \vec{U}].P} : Y - Y^S}\ \text{TC-INS2}$$

$$\frac{\Gamma, x^- : S, \vec{y} : \vec{U} \vdash_{X \cup Y^S - \{x^+\}} \boldsymbol{P} : Y \quad Y^S \subseteq Y \quad \vec{T} \leq \vec{U} \quad x^- \in X \cap Y}{\Gamma, x^- : ?[\vec{T}].S \vdash_X \boldsymbol{x?[\vec{y} : \vec{U}].P} : Y - Y^S}\ \text{TC-INS3}$$

$$\frac{(\Gamma, x : S) - Y^S \vdash_{X - Y^S} \boldsymbol{P} : Y \quad Y^S \subseteq X \quad \left(\Gamma, x : ![\vec{T}].S\right)[\vec{y}] \leq \vec{T} \quad x \in X \cap Y}{\Gamma, x : ![\vec{T}].S \vdash_X \boldsymbol{x![\vec{y}].P} : Y \cup Y^S}\ \text{TC-OUTS1}$$

$$\frac{(\Gamma, x^+ : S) - Y^S \vdash_{X - Y^S - \{x^-\}} \boldsymbol{P} : Y \quad Y^S \subseteq X \quad \left(\Gamma, x : ![\vec{T}].S\right)[\vec{y}] \leq \vec{T} \quad x^+ \in X \cap Y}{\Gamma, x^+ : ![\vec{T}].S \vdash_X \boldsymbol{x![\vec{y}].P} : Y \cup Y^S}\ \text{TC-OUTS2}$$

$$\frac{(\Gamma, x^- : S) - Y^S \vdash_{X - Y^S - \{x^+\}} \boldsymbol{P} : Y \quad Y^S \subseteq X \quad \left(\Gamma, x : ![\vec{T}].S\right)[\vec{y}] \leq \vec{T} \quad x^- \in X \cap Y}{\Gamma, x^- : ![\vec{T}].S \vdash_X \boldsymbol{x![\vec{y}].P} : Y \cup Y^S}\ \text{TC-OUTS3}$$

$$\frac{m \leq n \quad \forall i \in [m].\, (\Gamma, x : T_i \vdash_X \boldsymbol{P_i} : Y) \quad x \in X \cap Y}{\Gamma, x : \&\langle l_i : T_i \rangle_{i \in [m]} \vdash_X \boldsymbol{x \triangleright \{l_i : P_i\}_{i \in [n]}} : Y}\ \text{TC-OFFER1}$$

$$\frac{m \leq n \quad \forall i \in [m].\, \left(\Gamma, x^+ : T_i \vdash_{X - \{x^-\}} \boldsymbol{P_i} : Y\right) \quad x^+ \in X \cap Y}{\Gamma, x^+ : \&\langle l_i : T_i \rangle_{i \in [m]} \vdash_X \boldsymbol{x \triangleright \{l_i : P_i\}_{i \in [n]}} : Y}\ \text{TC-OFFER2}$$

$$\frac{m \leq n \quad \forall i \in [m].\, \left(\Gamma, x^- : T_i \vdash_{X - \{x^+\}} \boldsymbol{P_i} : Y\right) \quad x^- \in X \cap Y}{\Gamma, x^- : \&\langle l_i : T_i \rangle_{i \in [m]} \vdash_X \boldsymbol{x \triangleright \{l_i : P_i\}_{i \in [n]}} : Y}\ \text{TC-OFFER3}$$

$$\frac{\Gamma, x : T_i \vdash_X \boldsymbol{P} : Y \quad l = l_i \in \{l_1, ..., l_n\} \quad x \in X \cap Y}{\Gamma, x : \oplus\langle l_i : T_i \rangle_{i \in [n]} \vdash_X \boldsymbol{x \triangleleft l.P} : Y}\ \text{TC-CHOOSE1}$$

$$\frac{\Gamma, x^+ : T_i \vdash_{X - \{x^-\}} \boldsymbol{P} : Y \quad l = l_i \in \{l_1, ..., l_n\} \quad x^+ \in X \cap Y}{\Gamma, x^+ : \oplus\langle l_i : T_i \rangle_{i \in [n]} \vdash_X \boldsymbol{x \triangleleft l.P} : Y}\ \text{TC-CHOOSE2}$$

$$\frac{\Gamma, x^- : T_i \vdash_{X - \{x^+\}} \boldsymbol{P} : Y \quad l = l_i \in \{l_1, ..., l_n\} \quad x^- \in X \cap Y}{\Gamma, x^- : \oplus\langle l_i : T_i \rangle_{i \in [n]} \vdash_X \boldsymbol{x \triangleleft l.P} : Y}\ \text{TC-CHOOSE3}$$

Figure 6: Algorithmic Inference Rules for Typechecking

3.3 Operational Semantics

As specified in *Gay and Hole* [1], the language is evaluated via small-step operation semantics, where the program is rewritten at every step of the reduction. The reduction rules are specified as followed:

For reducing channel reads and writes on the same channel, where $\{z/y\}$ denotes substitution y with z:

$$\frac{}{x^p?[y:\ T].P \mid x^{\overline{p}}![z].Q \longrightarrow P\{z/y\} \mid Q}\ \text{R-Com}$$

For reducing branches and choices:

$$\frac{p \text{ is either } + \text{ or } - \quad 1 \leq i \leq n}{x^p \triangleright \{\ell_1 : P_1, \ldots, \ell_n : P_n\} \mid x^{\overline{p}} \triangleleft \ell_i.P \longrightarrow P_i \mid Q}\ \text{R-Select}$$

Reducing a process among processes executing in parallel:

$$\frac{P \longrightarrow P'}{P \mid Q \ \longrightarrow P' \mid Q}\ \text{R-Par}$$

Congruence rules can be applied during reduction:

$$\frac{P \cong\ P' \quad P \longrightarrow Q \quad Q \cong\ Q'}{P \longrightarrow Q'}\ \text{R-Cong}$$

For non-session-typed, *regular* channels, performing an IO action on the channel does not modify the channel type:

$$\frac{P \longrightarrow P' \quad T \text{ is not a session type}}{(\nu x:\ T)P \longrightarrow (\nu x:\ T)P'}\ \text{R-New}$$

For session-typed channels, performing an I/O action on the channel name consumes the head of the channel's session type (where the head is understood to be the next available operation for that name, according to the type syntax described in the previous section), creating a channel with a new type of tail(S):

$$\frac{P \longrightarrow P'}{(\nu x:\ S)P \longrightarrow (\nu x:\ \text{tail}(S))P'}\ \text{R-New}$$

4 Interpreter Implementation

4.1 Evaluation

We observed that there were a couple difficulties following the exact operational semantics specified by *Gay and Hole* [1] for programming an interpreter, particularly with replication. The reason replication is tricky is that we want to eagerly reduce processes, but lazily replicate. That is, we want to replicate a process only when all processes are blocked waiting for an IO action, and we want to replicate just the right processes that will unblock the current context. While it isn't too difficult to select the right process to replicate when we evaluate programs by hand, it is not trivial to come up with a general algorithmic approach to make such selection.

More specifically, for instance, consider the following scenario:

```
!((ν x: [!str. ...]) (x![...].y?[...]...) || (x?[...].z?[...]...))
```

where a replicated process begins with creating a (session-type) channel, forks parallel processes and executes some of its own IO action before communicating with any process that existed in the running context; in these cases, we have to look ahead multiple steps of a process before we decide to replicate, and potentially consider multiple replicate-able processes together. Moreover, the scoping issues with creating a new channel inside a replicating process further complicates this issue.

To that end, we modify the actual operational semantics of the language in the following ways:

1. Disallow channel creation at the head of replicating processes; our implementation explicitly rejects replication that begins immediately with a channel creation. We believe this restriction is not significant because, as expanded upon in the section on typing rules, no session-typed channels are available to replicating processes for reasons concerning the ownership of linear types. Therefore, the first communication between replicating processes and the outside context *must* happen through regular channel types. So (a) the creation of session-typed channels before the first communication is not essential and (b) any work preceding such communication and involving the creation of additional channels can be transferred to the process participating in the other end of the communication. This transfer should not be problematic as the other participant cannot be another replicating process - the heuristic of lazy replication requires replication to be on demand and, therefore, cannot occur between two processes awaiting replication.

2. Realize the following congruence rules:

$$!(P \mid Q) \cong\, !P \mid !Q$$
$$!(!P) \cong\, !P$$

   Before executing a program, we reduce every replication to "*IO Normal Form*" (IONF) using the above congruence rules. With this, any replication will begin with some IO action at execution. More specifically, since we disallow channel creation inside replication, the channels appearing at the beginning of the replication will be a *regular* (global) channel, meaning we can select which process to replicate just by looking at the head of the replicate-able processes.

With the modification above, we outline the way our interpreter executes programs as followed. At every step of the reduction, we rewrite the running context, defined as followed:

```
type context =
  { active: Process list  (* Processes currently executing in parallel *)
    reps: Process list    (* Replication processes *)
  }
```

First, we attempt to reduce processes in the `active` list according to the reduction rules defined in Section 3.3. When we see a replication for the first time (`!P`) in the `active` list, we remove `!P` from it, and add `P` to the `reps` list.

We attempt to reduce the `active` list until we are stuck (that is, every processes is waiting on some IO action). If we are stuck, we scan through the `reps` list to find a process that can unblock the current context; recall that, with our restriction, we only need to look at the head of each process to determine which process to replicate. If such process is found, add it to the `active` list and attempt reducing it again. If no such process is found, then the program is deadlocked.

4.2 Type System

The type system is implemented exactly as specified in *Gay and Hole* [1], and detailed in Section 3.2.

4.3 Implementation Details

Our interpreter was implemented with the `OCaml` language, and can be found at https://github.com/session-pi/session-pi.

4.4 Example Programs

We demonstrate the following examples. More rigorous testing can be found in our Github repository.

Example 1: Sending channels through other channels

```
(v x: RecType: T.[?[T].End])
    ((v y: ?[RecType: T.[?[T].End]].End)!x[y].!y[x].0)
    ||
    (?x[y: ?[RecType: T.[?[T].End]].End].?y[z: RecType: T.[?[T].End]].0)
```

The program above passes our type-checker, and executes as followed:

Evaluation

```
active[
  (v x: RecType: T.[?[T].End])
      ((v y: ?[RecType: T.[?[T].End]].End)!x[y].!y[x].0)
      ||
      (?x[y: ?[RecType: T.[?[T].End]].End].?y[z: RecType: T.[?[T].End]].0)
]
active[
  ((v y: ?[RecType: T.[?[T].End]].End)!x[y].!y[x].0)
  ||
  (?x[y: ?[RecType: T.[?[T].End]].End].?y[z: RecType: T.[?[T].End]].0)
]
active[
  (v y: ?[RecType: T.[?[T].End]].End)!x[y].!y[x].0,
  ?x[y: ?[RecType: T.[?[T].End]].End].?y[z: RecType: T.[?[T].End]].0
]
active[
  ?x[y: ?[RecType: T.[?[T].End]].End].?y[z: RecType: T.[?[T].End]].0,
  !x[y].!y[x].0
]
active[
  ?y[z: RecType: T.[?[T].End]].0,
  !y[x].0
]
active[
  0,
  0
]
```

Example 2

```
(v x : ^['[?{int, int}.!{int}.!{int}.end]])
   ((v y : '[?{int, int}.!{int}.!{int}.end])
    x![y].
    y![2, 3].
    y?[z : int].
    y?[w : int].
    zero
    |
    x?[y : '[?{int, int}.!{int}.!{int}.end]].
    y?[z : int, w : int].
    y![z].
    y![w].
    zero)
```

The program above passes type-checking (correctly), and evaluates as followed:

Example 2 Execution

```
active[
  (v x: [?[Int, Int].![Int].![Int].End])
    ((v y: ?[Int, Int].![Int].![Int].End)
      !x[y].!y[2, 3].?y[z: Int].?y[w: Int].0)
    ||
    (?x[y: ?[Int, Int].![Int].![Int].End].?y[z: Int, w: Int].!y[z].!y[w].0)
]
active[
  ((v y: ?[Int, Int].![Int].![Int].End)
    !x[y].!y[2, 3].?y[z: Int].?y[w: Int].0)
  ||
  (?x[y: ?[Int, Int].![Int].![Int].End].?y[z: Int, w: Int].!y[z].!y[w].0)
]
active[
  (v y: ?[Int, Int].![Int].![Int].End)
    !x[y].!y[2, 3].?y[z: Int].?y[w: Int].0,
  ?x[y: ?[Int, Int].![Int].![Int].End].?y[z: Int, w: Int].!y[z].!y[w].0
]
active[
  ?x[y: ?[Int, Int].![Int].![Int].End].?y[z: Int, w: Int].!y[z].!y[w].0,
  !x[y].!y[2, 3].?y[z: Int].?y[w: Int].0
]
active[
  ?y[z: Int, w: Int].!y[z].!y[w].0,
  !y[2, 3].?y[z: Int].?y[w: Int].0
]
active[
  !y[z].!y[w].0,
  ?y[z: Int].?y[w: Int].0
]
active[
  ?y[w: Int].0,
  !y[w].0
]
active[
  0,  0
]
```

Now, we will try to break this program:

Breaking Example 2

```
(v x : ^['[?{int, int}.!{int}.!{int}.end]])
   ((v y : '[?{int, int}.!{int}.!{int}.end])
    x![2].   -- incorrect!
    y![2, 3].
    y?[z : int].
    y?[w : int].
    zero
    |
    x?[y : '[?{int, int}.!{int}.!{int}.end]].
    y?[z : int, w : int].
    y![z].
    y![w].
    zero)

(v x : ^['[?{int, int}.!{int}.!{int}.end]])
   ((v y : '[?{int, int}.!{int}.!{int}.end])
    x![y].
    y![2].   -- incorrect!
    y?[z : int].
    y?[w : int].
    zero
    |
    x?[y : '[?{int, int}.!{int}.!{int}.end]].
    y?[z : int, w : int].
    y![z].
    y![w].
    zero)

(v x : ^['[?{int, int}.!{int}.!{int}.end]])
   ((v y : '[?{int, int}.!{int}.!{int}.end])
    x![y].
    y![2, false].   -- incorrect!
    y?[z : int].
    y?[w : int].
    zero
    |
    x?[y : '[?{int, int}.!{int}.!{int}.end]].
    y?[z : int, w : bool]. -- incorrect!
    y![z].
    y![w].
    zero)

(v x : ^['[?{int, int}.!{int}.!{int}.end]])
   ((v y : '[?{int, int}.!{int}.!{int}.end])
    x![y].
    y![2, 3].
    y?[z : int].
    y?[w : int].
    zero
    |
    x?[y : '[?{int, int}.!{int}.!{int}.end]].
    y?[z : int, w : int].
    y![z].
    zero)   -- incorrect!
```

The four programs above do not pass our type-checking, resulting in the following errors respectively:

Breaking Example 2 Type-Checker Results

```
(Failure "incompatible output type")
(Invalid_argument "length mismatch in zip_exn: 1 <> 2")
(Failure "incompatible output type")
(Failure "channel name not available or not used")
```

Finally, the example below demonstrates recursive types, branches and choices

Recursive Types

```
(v x : m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]])
    (!(x?[y : '[&<'a : !{m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]]}.end,
                  'b : !{m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]],
                         m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]]}.end>]].
       y |> {'a : y![x].zero,
             'b : y![x, x].zero})
     |
     (v y : '[+<'a : ?{m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]]}.end,
                'b : ?{m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]],
                       m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]]}.end>])
     x![y].
     y <| 'a.
     y?[z : m T.^['[&<'a : !{T}.end, 'b : !{T, T}.end>]]].
     zero
    )
```

and evaluates as followed:

Recursive Types

```
active[
    (v x: RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}])
        (Rep (?x[y: &{'a: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
                      'b: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                            RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End}].
              y |> {'a: !y[x].0,
                    'b: !y[x, x].0}))
        ||
        ((v y: +{'a: ?[RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
                 'b: ?[RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                       RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End})
         !x[y].
         y <| 'a.
         ?y[z: RecType: T.[&{'a: ![T].End,
                             'b: ![T, T].End}]].0)
]
active[
    (Rep (?x[y: &{'a: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
                  'b: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                        RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End}].
          y |> {'a: !y[x].0,
                'b: !y[x, x].0}))
    ||
    ((v y: +{'a: ?[RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
             'b: ?[RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                   RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End})
     !x[y].
     y <| 'a.
     ?y[z: RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].0)
]
active[
    Rep (?x[y: &{'a: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
                'b: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                      RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End}].
         y |> {'a: !y[x].0,
               'b: !y[x, x].0})
    ,
    (v y: +{'a: ?[RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
            'b: ?[RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                  RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End})
    !x[y].
    y <| 'a.
    ?y[z: RecType: T.[&{'a: ![T].End,
                        'b: ![T, T].End}]].0
]
active[
  !x[y].y <| 'a.
  ?y[z: RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].0
]
active[
  !x[y].
  y <| 'a.
  ?y[z: RecType: T.[&{'a: ![T].End,
                      'b: ![T, T].End}]].0
  ,
  ?x[y: &{'a: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End,
          'b: ![RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}],
                RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].End}].
  y |> {'a: !y[x].0,
        'b: !y[x, x].0}
]
active[
  y |> {'a: !y[x].0,
        'b: !y[x, x].0}
  ,
  y <| 'a.
  ?y[z: RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].0
]
active[
  !y[x].0
  ,
  ?y[z: RecType: T.[&{'a: ![T].End, 'b: ![T, T].End}]].0
]
active[
  0, 0
]
```

5 Conclusion

For this project, we implemented a session-type-checker and an interpreter for the session-typed *pi* calculus. We demonstrate through this that the correctness of communication protocols of programs can be statically checked via a type system.

We thank John Hui, our project advisor, for his helpful insights.

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`.