A Session Type System with Subject Reduction

Keigo IMAI\textsuperscript{(a)}, Shoji YUEN\textsuperscript{(b)}, and Kiyoshi AGUSA\textsuperscript{(c)}, Members

SUMMARY Distributed applications and services have become pervasive in our society due to the widespread use of internet and mobile devices. There are urgent demands to efficiently ensure safety and correctness of such software. A session-type system is a framework to statically check whether communication descriptions conform to certain protocols. They are shown to be effective yet simple enough to fit in harmony with existing programming languages. In the original session type system, the subject reduction property does not hold. This paper establishes a conservative extension of the original session type system with the subject reduction property. Finally, it is also shown that our typing rule properly extends the set of typeable processes.

key words: Pi-calculus, session types, type system, subject reduction, concurrency, process calculi, polarity

1. Introduction

One of the most basic and important properties of distributed computing is communication safety, meaning that there is no mismatch between inputs and outputs. The session type system \cite{1} has been proposed to ensure communication safety. Thus it provides a good basis to syntactically analyze, specify and check communication protocols in software. The theory of session types is originally developed for a version of the \pi-calculus \cite{2}. It has been shown to be effective yet simple enough to fit existing programming languages \cite{3,4}. Session types have also been implemented in several programming languages \cite{5,7,8} including Java, C\#, Haskell, ML and Scala. Our current goal is to reinforce the theoretical basis of type system for communication safety.

For example, the following session type describes the protocol of the remote procedure call that receives an integer, then sends an integer as the return value,

\[ ?\text{int}!\text{int};\text{end} \]

where ?\text{int}; and !\text{int}; denote receiving and sending of an integer, respectively, and \text{end} is a termination. In the following process the above type is assigned to \text{rpc}. The process returns the doubled value of the received integer \(x\).

\[ \text{rpc}(x); \text{rpc}(x + 2); 0 \]

(2)

\(s?();\) and \(s!(\cdot)\) are receiving and sending on the channel \(s\), respectively, and \(0\) is a terminated process.

When a communication involves channel-passing, the subject reduction property does not hold \cite{9}. See the following extension of the above RPC server.

\[ \text{init}(!\text{dbg}; \text{rpc}(x)); \text{rpc}(x); \text{dbg}(x); \text{dbg}(y); 0 \]

(3)

The prefix \text{init}(!\text{dbg}); receives a channel on \text{init} then binds it to \text{dbg}. \(P|Q\) denotes that \(P\) and \(Q\) are executed in parallel. In summary, (3) receives a channel on \text{init} for initialization, then receives the argument of RPC on \text{rpc}. It returns the result on \text{rpc}, and in parallel, sends the argument on \text{dbg} (possibly for some debugging purpose), then receives a value on \text{dbg}. Process (3) is well-typed by assigning the following type to \text{init} and type (1) to \text{rpc},

\[ ?[\text{int}];!\text{int};\text{end};\text{end} \]

(4)

where ?[\text{T}]; denotes a protocol that receives a channel with type \text{T}. Bound channel \text{dbg} has type ?[\text{int}];!\text{int};\text{end}.

Let us consider that channel \text{rpc} is sent to (3) through \text{init} by some other process in the environment. Firstly, in that process the type of the channel \text{rpc} is required to be

\[ !\text{int};!\text{int};\text{end} \]

(5)

which denotes the client side’s protocol of the RPC. Check that (5) is reciprocal or dual (the protocol on the opposite side) to the server’s side (1). Since it is identical to the type inside \((\cdot)\) of (4) \text{rpc} can be communicated through \text{init}.

By that communication step, the prefix is removed from (3) and \text{dbg} is substituted with \text{rpc}, as follows.

\[ \text{rpc}(x); \text{rpc}(x + 2); 0 \mid \text{rpc}(x); \text{rpc}(y); 0 \]

(6)

Now, the original session type system \cite{1} somehow rejects this as untypeable, and subject reduction fails. The failure just comes from a syntactic mismatch between protocols of both sides of | (the underlined parts). The left hand side of | sends an integer, while the right hand side sends a integer then receives an integer.

We observe that both (3) and (6) are typeable. (3) works well when a channel other than \text{rpc} is received. When \text{rpc} is delivered via \text{init}, the communication safety is literally kept in (6), since it is deadlocked. We consider processes
deadlocked if they do not evolve further, but not structurally equivalent to \( 0 \). Note that (6) cannot communicate on \( \text{rpc} \) further, since both client’s and server’s side of \( \text{rpcs} \) are now held by (6). Detection of deadlock requires much more complex typing discipline [4], [10], [11], whereas our objective is not to obtain deadlock-freeness, but a good compromise between safety and usability.

From the observation above, we show that the subject reduction property is established in our extension \( \mathbf{R} \) of the original session type system [1]. The key idea is to intentionally give a special type \( \text{Odd} \) which denotes a mismatch of protocols. For example, channel \( s \) in (6) has type \( \text{int};\text{Odd} \), which says that after sending an integer on \( s \), the process may go into an error state. And our subject reduction theorem states that all session types in \( \mathbf{R} \), including \( \text{int};\text{Odd} \), the type above, will never become \( \text{Odd} \) i.e. never reach an error.

In summary, we construct a type judgment \( \vdash_R \) such that

1. If \( \vdash P \) in [1] then \( \vdash_R P \).
2. If \( \vdash_R P \) then \( P \) is safe.
3. Subject reduction holds, i.e. if \( \vdash_R P \) and \( P \rightarrow P' \) then \( \vdash_R P' \).

and we obtain for all \( P \) s.t. \( \vdash P, P \) always reduces to a safe process, by an obvious induction on the length of reduction.

The rest of this paper is organized as follows. In Sect. 2, session types and the \( \pi \)-calculus are introduced from [1]. In Sect. 3, we prove the failure of subject reduction in the original session type system [1] in depth. In Sect. 4, we propose an extended type system \( \mathbf{R} \) in that subject reduction holds. In Sect. 5, we prove subject reduction, type safety, and related properties of \( \mathbf{R} \). Section 6 describes related work and Sect. 7 concludes this paper.

2. The Session Type System in the \( \pi \)-Calculus [1], [9]

In this section we review the binary session type system for the \( \pi \)-calculus. All definitions are brought from [9]. We omit the syntax for session acceptance and session request to initialize a session, since we are only interested in channel-passing in session-typed communications. Furthermore we exclude recursion for simplicity.

2.1 \( \pi \)-Calculus with Labels and Values

A session is a sequence of communications on a channel shared between processes. In session-typed \( \pi \)-calculus, three kinds of messages, basic values, labels and channels, are communicated in such sessions. A channel is ranged over by \( s, s', t, t', u, u', r, r', \ldots \), used by processes to communicate privately with each other, and can be passed through another channel. A label is either left or right, used to indicate branches in a session. A value includes \{true, false\} and is ranged over by \( v \). The syntax of an expression \( e \) is left abstract, and an evaluation relation \( e \Downarrow v \) is assumed. A countably infinite set of variables, ranged over by \( x, y, \) are disjoint from the set of channels. Variables only occurs in \( e \).

### Definition 1 (Processes)

\[
P \equiv Q \text{ if } P \equiv_a Q \quad P \mid Q \equiv (P | Q)\quad (P | Q) \equiv (P | (Q | R)) \quad ((P | Q)) \equiv (P | (Q | R))
\]

\[
\begin{align*}
\text{if } & e \text{ is } \text{left} \quad & (\alpha) \text{ if } & e \text{ is } \text{right} \\
\text{and } & e \text{ is } \text{left} \quad & (\alpha) \text{ if } & e \text{ is } \text{right} \\
\text{and } & e \text{ is } \text{left} \quad & (\alpha) \text{ if } & e \text{ is } \text{right} \\
\text{and } & e \text{ is } \text{left} \quad & (\alpha) \text{ if } & e \text{ is } \text{right} \\
& e \text{ is } \text{left} \quad & (\alpha) \text{ if } & e \text{ is } \text{right} \\
\end{align*}
\]

### Definition 2 (Prefix and subject)

A process \( P \) is prefixed or a prefix if \( P \) is either of the form \( s!(e); P, s![](left); P, s![](right); P \) or \( s!(e); P \). In other words, \( 0, (vs)P \) and \( P|Q \) are not prefixed. We call channel \( s \) in them as the subject of the prefix.

In \( \pi \)-calculus structurally equivalent terms like \( P|Q \) and \( Q|P \) are identified with each other. Renaming of bound variables and channels is called \( \alpha \)-conversion as usual. We denote \( P \equiv_{\alpha} Q \) if \( Q \) is an \( \alpha \)-conversion of \( P \) and vice versa, and identify them.

### Definition 3 (Structural congruence)

The structural congruence is the smallest relation that satisfies the axioms in Fig. 1.

### Definition 4 (Reduction relation)

The reduction relation \( \rightarrow \) is inductively defined by the rules in Fig. 2. \( \rightarrow^* \) is the reflexive transitive closure of \( \rightarrow \). We write \( P \not\rightarrow \) if there is no \( P' \) s.t. \( P \rightarrow P' \).

A mismatch of message kind in a communication is called an error, and rejected by the session type system. Errors are formalized using the following notions.

### Definition 5 (s-process and s-redex)

A \( s \)-process is a closed and prefixed process whose subject is \( s \). A \( s \)-redex

\[\text{Under the session type discipline, (vs) hardly has any semantic impact on the operational semantics since typing rule [CONC] (see Sect. 2.2) restricts more than two parallel use of channels.}\]
is a parallel composition of a reducible pair of $s$-processes, namely either of form $s!(e);P_1 \mid s?[(x);P_2; s \triangleright \text{left}; P \mid s \triangleright \text{right}; P \mid s \triangleright \text{left}; P_1, \text{right}; P_2]$, $s \triangleright \text{right}; P \mid s \triangleright \text{left}; P_1, \text{right}; P_2$ or $s!(\ell);P \mid s?[(\ell');Q \rightarrow P \mid Q | Q \rightarrow P' \mid Q \rightarrow P' | P \rightarrow P']$. $s$-redex.

Intuitively, a $s$-redex is a pair of processes that can communicate with each other on $s$, without any mismatch of message kinds.

**Definition 6** (Error): A process $P$ is an **error** if and only if there exist two $s$-processes $Q_1$ and $Q_2$ such that $P = (v \exists)(Q_1 \mid Q_2)0$ and $Q_1|Q_2$ is not an $s$-redex.

Note that a deadlock is **not** treated as an error.

2.2 The Session Type System and Type Safety

The session type system statically guarantees absence of errors by tracking every use of channels. A session type denotes the usage of a channel during a session. We assume a set of sorts of values including $[\text{bool}]$ ranged over by $S$.

**Definition 7** (Session types): The syntax of types or **session types** is defined by the following grammar:

$$T, T', T_1, T_2 ::= \backslash S : T \mid ?S : T \mid \emptyset \left[\text{left:} T_1, \text{right:} T_2\right] \mid \land \left[\text{left:} T_1, \text{right:} T_2\right] \mid \land \left[\text{left:} T_1 \mid \text{right:} T_2\right]$$ (8)

$\backslash S : T$; and $?S : T$; denotes communication of values of sort $S$, whereas $\left[\text{T} \mid \text{T}'\right]$; and $?\left[\text{T} \mid \text{T}'\right]$; are that of channels of type $T$. $\emptyset \left[\text{left:} T_1, \text{right:} T_2\right] \text{ and } \land \left[\text{left:} T_1, \text{right:} T_2\right]$ are selection and offer, respectively. The above-mentioned types are called **prefixed**, without confusion with that for processes. end and $\bot$ are not prefixed. Prefixed types denote protocols of channels to communicate with the environment, whereas $\bot$ is given to channels that is used internally. end denotes a terminated session. A **dual** of a session type denotes the session of the other end of the channel.

**Definition 8** (Dual): The dual of a session type $\overline{T}$ is inductively defined by the following equation,
3. Subject Reduction Failure

We elaborate the failure of subject reduction in the original session type system [1] for further development. The failure does not mean loss of communication safety, and from that, we motivate the need for reformulation of typing rules in [1].

We show three aspects of failure of subject reduction. All cases are derived by a particular case of channel-passing (Pass of Fig. 2),

\[ s!(t); P | s?((u);Q) \leadsto P | Q[t/u] \]  

where \( t \in \text{fc}(Q) \) and \( Q[t/u] \) is not typeable. The untypeability comes from three rules [Conc], [CRRes] and [Thr] of Fig. 3.

3.1 A Type Change During Reduction

All kinds of failure of subject reduction involves change of a type during reduction. The ‘subject reduction’ for a type system means that all types do not change during reduction in general, and also in particular for the original type system [1], even though type change do not mean loss of communication safety.

The typical case is a process called forwader that forwards a value from \( s \) to \( t \), as follows.

\[ \vdash s?; x!; x; 0 \triangleright s : ?S; \text{end} : t : !S; \text{end} \]  

(12)

There can be a case that \( t \) is substituted with \( s \) by Pass rule. Then, the type of \( s \) changes because of that substitution, i.e.

\[ \vdash s?; x!; x; 0 \triangleright s : ?S; !S; \text{end} \]  

(13)

This particular case is firstly pointed out by [9]. The type change implies a deadlock like (6) in Introduction, though communication safety is still kept i.e. (13) is not an error. However, such type change are considered wrong in the original session type system, and that view is also reflected in rule [CRRes].

Example 1: Let \( P_{e1} \) be

\[ *(ys)(u!(s)); 0 | u!(s); s?(x); x!; 0 \]

(14)

See that \( \vdash P_{e1} \triangleright u : \bot \) whereas the reduction of \( P_{e1} \) result in

\[ *(ys)s?(x); s!(x); 0 \]  

(15)

which is not typeable. Check that rule [CRRes] rejects (15), because the type of \( s \) of \( s!(y); s!(y); 0 \) is not \( \bot \) but \( ?S; !S; \text{end} \) for some \( S \).

See that type change destroys typeability of a term, since rule [CRRes] does not allow any types other than \( \bot \).

3.2 Detection of Unreachable Error

The next case involves mismatch of protocol between the two sides of [ ], as we have seen in RPC server process (6) in Introduction. The following example explains its typing in detail.

Example 2: Let \( P_{e2} \) be

\[ u!(t); s!(true); (s?(x); 0 | t?(y); t!(y); 0) | u!(s); 0 \]  

(16)

We have \( \vdash P_{e2} \triangleright u : \bot, s : \bot \). A reduction \( P_{e2} \leadsto P_{e2}' \) exists where \( P_{e2}' \) is

\[ s!(true); (s?(x); s!(y); 0) \]  

(17)

and \( \not\vdash P_{e2}' \) since the direct subterm of \( P_{e2}' \) is not typeable, i.e. \( \not\vdash s?(x); 0 | s!(y); s!(y); 0 \).

The non-typeability comes from the fact that [Conc] rules out parallel composition of two incompatible terms in the subterm of (17). However, note that (17) is not an error, since that incompatible part is prefixed by \( s!(true); \).

3.3 Limitation on Typing Rule for Delegation

The other cause of failure is limitation of [Thr] rule that gives types to process \( s!(t); P \). One is the form of \( s!(s); P \) that is not typeable in the usual session type systems. The following is rather subtle and seldom gives intuitive meaning nor interesting application, but we introduce it since it is a minimal example that reduces to a process containing \( s!(s); P \).
Odd typically gives types to error processes. The session type system \[1\] is right hand side is indeed typeable. On the left hand side of (18), its typing is such that

\[
\forall \Delta, \Delta' \in T, \quad P \mapsto \Delta \quad \text{if} \quad s'(\langle x \rangle; r') end \quad \text{if} \quad r P \mapsto \Delta' \quad \text{if} \quad s \not\equiv \Delta(s) \neq \text{Odd}, \text{then for all } P \mapsto r P' \mapsto \Delta' \quad \text{and} \quad \forall s, \Delta(s) \neq \text{Odd} \quad \text{hold.}
\]

Definition 11 (Odd type and threadiness): The set of session types in D is extended from the one in Definition 9 by the following grammar:

\[
T, T', T_1, T_2 := \cdots \mid \text{Odd}
\]

Odd is not a prefixed type, and \(\text{OddD}\) is not defined. We call T threaded if T is defined. In other words, T is threaded if it does not contain any \(\perp\) or \(\text{OddD}\).

Accordingly, the definition of composition and compatibility of typings is relaxed as follows:

Definition 12 (Extension of session type composition): The compatibility relation of D is \(\equiv_D\) such that \(\Delta_0 \equiv_D \Delta_1\) iff both \(\Delta_0(s)\) and \(\Delta_1(s)\) are threaded for all \(s \in \text{dom}(\Delta_0) \cap \text{dom}(\Delta_1)\). The composition of typings in D is written as \(\circ_D\) and defined by the following equation:

\[
(\Delta_0 \circ_D \Delta_1)(s) = \begin{cases} \perp & \text{if } s \in \text{dom}(\Delta_0) \cap \text{dom}(\Delta_1) \\ \Delta_0(s) & \text{if } s \in \text{dom}(\Delta_0) \cap \text{dom}(\Delta_1) \\ \Delta_1(s) & \text{if } s \in \text{dom}(\Delta_1) \setminus \text{dom}(\Delta_0) \end{cases}
\]

The intention behind introducing Odd is to mark the two concurrent usages of a channel as an error if two use of a channel is not dual with each other. For example, the subterm of (17), \(s'(\langle y \rangle; r') \mid s'(\langle y \rangle; r')\) is typeable under D, and Odd is assigned to s.

Then we later prove subject reduction (Theorem 1) which states that the types of all channels will not become Odd if they are not Odd, meaning that typeable processes never reach an error state. In other words, if \(r P \mapsto \Delta\) and \(\forall s, \Delta(s) \neq \text{Odd, then for all } P \mapsto r P' \mapsto \Delta' \quad \text{and} \quad \forall s, \Delta(s) \neq \text{Odd} \quad \text{hold.}
\]

Definition 11 (Odd type and threadiness): The set of session types in D is extended from the one in Definition 9 by the following grammar:

\[
T, T', T_1, T_2 := \cdots \mid \text{Odd}
\]

Odd is not a prefixed type, and \(\text{OddD}\) is not defined. We call T threaded if T is defined. In other words, T is threaded if it does not contain any \(\perp\) or \(\text{OddD}\).

Accordingly, the definition of composition and compatibility of typings is relaxed as follows:

Definition 12 (Extension of session type composition): The compatibility relation of D is \(\equiv_D\) such that \(\Delta_0 \equiv_D \Delta_1\) iff both \(\Delta_0(s)\) and \(\Delta_1(s)\) are threaded for all \(s \in \text{dom}(\Delta_0) \cap \text{dom}(\Delta_1)\). The composition of typings in D is written as \(\circ_D\) and defined by the following equation:

\[
(\Delta_0 \circ_D \Delta_1)(s) = \begin{cases} \perp & \text{if } s \in \text{dom}(\Delta_0) \cap \text{dom}(\Delta_1) \\ \Delta_0(s) & \text{if } s \in \text{dom}(\Delta_0) \cap \text{dom}(\Delta_1) \\ \Delta_1(s) & \text{if } s \in \text{dom}(\Delta_1) \setminus \text{dom}(\Delta_0) \end{cases}
\]

4. Reformulation of Typing Rules

We construct a session type system R and its typing judgment \(\tau_R\) in that subject reduction holds i.e. never fail into error nor untypeable. In other words, for all P if \(\tau_R P \mapsto \Delta\), then P is not an error, and for all \(P'\) s.t. \(P \mapsto r P' \mapsto \Delta'\) for some \(\Delta'\). Allowance of type change is one key point, but it is not sufficient as we have seen in the examples in the previous section.

4.1 The Odd Type and Relaxation of Typing Compatibility

Firstly, we establish a type system D (Danger) that intentionally gives types to error processes. The session type Odd is introduced to denote an error on that channel. For example, the subterm of (17), \(s'(\langle y \rangle; r') \mid s'(\langle y \rangle; r')\) is typeable under D, and Odd is assigned to s.

The intention behind introducing Odd is to mark the two concurrent usages of a channel as an error if two use of a channel is not dual with each other. For example, the subterm of (17), \(s'(\langle y \rangle; r') \mid s'(\langle y \rangle; r')\) is typeable under D, and Odd is assigned to s. Thus errors are ensured not to be reached.

Proposition 1: The following holds:

1. For all \(\Delta_1\) and \(\Delta_2\), if \(\Delta_1 \not\equiv_D \Delta_2\) then \(\Delta_1 \not\equiv_D \Delta_2\).
2. For all \(\Delta_1\) and \(\Delta_2\), if \(\Delta_1 \not\equiv_D \Delta_2\) then \(\Delta_1 \circ_D \Delta_2 = \Delta_1 \circ_D \Delta_2\).

\[
\begin{array}{c}
\frac{u?\langle x \rangle; r!(\langle x \rangle); 0 \mapsto \text{end} - u : ?S\cdot \text{end} \cdot r : !S\cdot \text{end}}{\text{by Tim}} \\
\frac{u!(\langle x \rangle; u?\langle x \rangle; r!(\langle x \rangle); 0 \mapsto \text{end} - s : ?S\cdot \text{end} \cdot u : ?S\cdot \text{end} \cdot r : !S\cdot \text{end}}{\text{by Car}} \\
\frac{\sigma?\langle x \rangle; u!(\langle x \rangle; u?\langle x \rangle; r!(\langle x \rangle); 0 \mapsto \text{end} - s : ?S\cdot \text{end} \cdot u : ?S\cdot \text{end} \cdot \text{end} \cdot s : ?S\cdot \text{end} \cdot !S\cdot \text{end}}{\text{by Car}} \\
\end{array}
\]
4.2 Typing Rule Replacements

We introduce a new judgment $\Gamma \vdash_D P \rightarrow \Delta$ of D.

**Definition 13** (Type system D): The type judgment $\Gamma \vdash_D P \rightarrow \Delta$ is inductively defined by the rules in Fig. 3 except for [Conc], [CRes] and [Thr], and including [Ex-Conc], [Ex-CRes] and [Ex-Thr] in Fig. 5.

The replacement of three rules corresponds to the three modes of failure introduced in Sect. 3.

[Ex-Conc] relaxes the compatibility condition to $\equiv_D$, which intentionally allows incompatible or unsafe use of channels. If $(\Delta_1 \circ_D \Delta_2)(s) = \text{Odd}$ holds, it indicates that there’s some possibility of an error on $s$ in $P$. (In R, such non-prefixed occurrence of Odd is rejected.) In addition to that, a channel with prefixed Odd (like `bool;Odd) will never be used since Odd has no dual type. This prevents types to become Odd, ensuring that processes will not become an error.

[Ex-CRes] relaxes [CRes] by allowing $s$ to also have a prefixed type to give type to terms like $[T]$ by a single rule.

[Ex-Thr] is the novel typing rule which gives types to more diverse styles of delegation than [Thr] by a single rule. Literally, $\Delta \circ_D t : T'$ in premise is read as

after delegation of channel $t$, $t$ will be used at the partner’s place according to $T'$ in parallel with $\Delta$

Remember that $\circ$ (and $\circ_D$) means a parallel composition of typings. The key is the use of $\circ_D$ which allows $t \in \text{dom}(\Delta)$, differently from [Thr]. And $\Delta \circ_D t : T' = \Delta' \cdot s : T$ is read as

after delegation, $s$ is used according to $T$ at somewhere

Here the novelty lies in that it covers $s = t$, in that $s(!\langle s \rangle) : P$ is allowed and after delegation $s$ is used at the partner’s place. Then the typing $s : ![T'] ; T$ in conclusion holds, which says

sending of a channel of type $T'$ on $s$ will have protocol $![T'] ; T$.

The rationale behind that is explained as follows. Firstly, assume $s \neq t$ and $t \notin \text{dom}(\Delta)$. In this case the rule is identical to [Thr]. Since $\Delta \circ_D t : T' = \Delta \cdot t : T'$ we obtain $\Delta \cdot t : T' = \Delta' \cdot s : T$ in premise. By picking some $\Delta''$ which satisfies $\Delta \cdot t : T' = \Delta' \cdot s : T = \Delta'' \cdot s : T \cdot t : T'$, the rule becomes identical to [Thr] rule.

Next, consider the case $s \neq t$ and $t \in \text{dom}(\Delta)$ (hence $t \in \text{fc}(P)$). It allows terms like $(\pi t)! \langle t \rangle : P$ in that $t$ occurs in $P$, usually called bound output. Such process is used to send a private channel $t$ via $s$, ensuring that channel $t$ is local to $P$. We put identical rule [Pseudo-Ex-Thr] in Fig. 6. This case is not our invention, and similar typing rule can also be seen in a few session type papers [12]–[14]. Let us check that this case is actually identical to [Pseudo-Ex-Thr]$.\Delta$. Let $\Delta = \Delta' \cdot s : T \cdot t : T_1$. Note that if $t : T_1 \nvdash_D t : T'$, $\Delta \circ_D t : T' = \Delta' \cdot s : T \cdot t : T_1 \circ_D t : T' = \Delta'' \cdot s : T \cdot t : T_2$ for some $T_2$, and from that and $\Delta \circ_D t : T_1 = \Delta' \cdot s : T$ we obtain $\Delta' = \Delta'' \cdot t : T_2$. $T_2$ can be $\bot$ or Odd depending on whether $T_1$ and $T'$ are dual of each other or not. If $\bot$ is the type, we can expect that there’s no error in further communication on $t$, since the receiver of $t$ uses that channel according to $T'$, at the same time $P$ uses $t$ according to $T''$.

Then the case $s = t$ is divided into the two sub-cases, depending on $s \notin \text{dom}(\Delta)$ or not. When $s \in \text{dom}(\Delta)$ holds, we can conclude that it is deadlockened in any way. See rule [Pseudo-Ex-Thr] in Fig. 6 which is obtained by imposing $s = t \land s \in \text{dom}(\Delta)$ on [Ex-Thr]. We have either $T = \text{Odd}$ or $T = \bot$. It is deadlockened and would not reduce to an error, since it is prefixed by $s$ and any other process can not have $s$, because of the fact that there’s no dual of $\bot$ and $\text{Odd}$.

Lastly, we get some interesting processes from the final case. Assuming $s = t$ and $s \notin \text{dom}(\Delta)$ in [Ex-Thr] (i.e. $P$ does not use $s$ anymore), we get the rule [Pseudo-Ex-Thr] in Fig. 6. Unlike [Pseudo-Ex-Thr], the processes typeable here are not deadlockened and can communicate with the environment on $s$. The type $\pi [T] ; T$ says that “after delegating a channel that has type $T$, $s$ is used according to $T''$, even though $P$ never use $s$, but the receiver of $s$ uses it in the rest of the process. The rationale of this case is following. Consider that a judgment

$\Gamma \vdash_D s ! \langle \langle s \rangle \rangle ; P | s ? \langle \langle i \rangle \rangle ; Q \rightarrow \Delta \cdot s : \bot$

where $s \notin \text{fc}(P)$. Here, $s ! \langle \langle s \rangle \rangle ; P$ delegates the rest of the usage of $s$ via $s$ itself. Moreover, since we have $\Gamma \vdash_D Q \rightarrow \Delta' \cdot s : T \cdot t : T$ for some $\Delta'$, after a reduction we can expect that $Q[s/t]$ may further communicate on $s$. An example of reduction is following:

\[
\begin{align*}
\vdash s ! \langle \langle s \rangle \rangle ; 0 | s ? \langle \langle i \rangle \rangle ; s ! \langle \langle \text{true} \rangle \rangle ; 0 | t ? \langle \langle x \rangle \rangle ; 0 \rightarrow s ! \langle \langle \text{true} \rangle \rangle ; 0 | s ? \langle \langle x \rangle \rangle ; 0 \rightarrow 0
\end{align*}
\]

(21)

It is even novel since many standard type systems like i/o-types [15] or linear types [16] can not give a type to $s ! \langle \langle s \rangle \rangle ; P$ unless recursive typing is introduced$^7$.

**Definition 14** (The session type system R): The type judgment $\Gamma \vdash_R P \rightarrow \Delta$ is defined by $\Gamma \vdash_R P \rightarrow \Delta$ if $\Gamma \vdash_D P \rightarrow \Delta$ and

$^7$The sorting system of the polyadic $\pi$-calculus by Milner [17] does allow $s ! \langle \langle s \rangle \rangle ; P$, however, it can only be formulated in a type system with recursive types.
\[ \forall s, \Delta(s) \neq \text{Odd.} \]

\( R \) excludes errors by prohibiting Odd in typing.

5. Subject Reduction and Type Safety

5.1 The Basic Properties of the Type System

Firstly, we prove a few basic lemmas for \( \vdash_D \) and \( \vdash_R \).

**Lemma 1 (Weakening):** Let \( \Gamma \vdash_D P \Rightarrow \Delta \).

1. If \( x \notin \text{fv}(P) \), then \( \Gamma \setminus x : v \vdash_D P \Rightarrow \Delta \).
2. If \( s \notin \text{fc}(P) \), then \( \Gamma \vdash_D P \Rightarrow \Delta \cdot s : T \) where \( T = \text{end} \) or \( T = \bot \).

The same holds for \( \vdash_R \).

**Proof:** By simple induction on the derivation of the typing judgment. For 2, the base case is \([\text{In}xct]\) and use \([\text{Bor}]\) on the derivation if \( T = \bot \). Note that in \( \vdash_D (\forall t)P \Rightarrow \Delta \) and \( s?([t])P \Rightarrow \Delta, \Delta \) can contain \( t \) since we identify \( \alpha \)-convertible terms, e.g. \( (\forall t)P \) and \( (\forall t)(P[t/i]) \) are identified for some \( t' \notin \text{fc}(P) \). \( \square \)

**Lemma 2 (Strengthening):** Let \( \Gamma \vdash_D P \Rightarrow \Delta \).

- If \( x \notin \text{fv}(P) \), then \( \Gamma \setminus x : v \vdash_D P \Rightarrow \Delta \).
- If \( s \notin \text{fc}(P) \), then \( \Gamma \vdash_D P \Rightarrow \Delta \cdot s \).

The same holds for \( \vdash_R \).

**Proof:** By simple induction on the derivation of the typing judgment. \( \square \)

Usual inversion lemma of a type system that states the typeability of the direct subterm also holds for \( \vdash_D \) and \( \vdash_R \). However some trick is required to obtain the correct typing. This is because one can introduce arbitrary many occurrences of \( s : \bot \) into typings for any fresh \( s \) by using the rules \([\text{Bor}]\) and \([\text{Ena}]\). We relate them with the following partial order \( < \).

**Definition 15** ([9]): \( < \) is the partial order such that

\[ \Delta \cdot s : \text{end} \land \Delta' \cdot s : \bot \iff \Delta < \Delta'. \]  

\( \square \)

**Lemma 3 (Inversion):** The following holds:

1. If \( \Gamma \vdash_D P \vdash_R P_2 \Rightarrow \Delta \), then there exist \( \Delta_1, \Delta_2 \) such that \( \Gamma \vdash_D P_1 \Rightarrow \Delta_1, \Gamma \vdash_D P_2 \Rightarrow \Delta_2 \) and \( \Delta_1 \circ_D \Delta_2 < \Delta \).
2. If \( \Gamma \vdash_D (\forall s)P \Rightarrow \Delta \), then there exist \( \Delta', T \) such that \( \Delta' < \Delta \) and \( \Gamma \vdash_D P \Rightarrow \Delta' \cdot s : T \).

The same holds for \( \vdash_R \).

**Proof:** (1) The derivation tree of \( \Gamma \vdash_D P \vdash_R P_2 \Rightarrow \Delta \) has the [Conc] node rooted by the sequence of \([\text{Bor}]\) nodes, hence \( \Delta_1 \circ_D \Delta_2 < \Delta \). For \( \vdash_R \), the proof goes similar. Note that by definition we have \( \Delta(s) \neq \text{Odd} \) for all \( s \). Obviously \( (\Delta \circ_D \Delta_2)(s) \neq \text{Odd} \) also holds.

Furthermore, from the definition of \( \circ_D \) we can derive \( \Delta(s) \neq \text{Odd} \) for \( i \in \{1, 2\} \), hence \( \Gamma \vdash_R P_i \Rightarrow \Delta_i \) hold. (2) Trivially holds according to rule \([\text{Ex-CRes}]\). \( \square \)

Furthermore, we prepare Inversion lemma for \([\text{Ex-Thr}]\) for later use since \([\text{Ex-Thr}]\) is much sophisticated than the other rules. Intuitively, this lemma is that for cases of \([\text{Thr}]\), \([\text{Pseudo-Ex-Thr}_1]\), \([\text{Pseudo-Ex-Thr}_2]\) and \([\text{Pseudo-Ex-Thr}_3]\).

**Lemma 4** (Inversion for delegator): The following holds:

1. If \( \Gamma \vdash_D s([t]); P \Rightarrow \Delta \cdot s : T_1 \cdot t : T_1 \) and both \( T_2 \) and \( T_3 \) are threaded, then for some threaded \( T_1' \), \( T_3 = ![T_1]; T_3' \) and \( \Gamma \vdash_D P \Rightarrow \Delta' \cdot s : T_1' \cdot T_3' \) for some \( \Delta' < \Delta \).

2. If \( \Gamma \vdash_D s([t]); P \Rightarrow \Delta \cdot s : T' \cdot T_1 \) and \( T_3 \) is threaded while \( T' \) is not, then \( T_3 = ![T_1]; T_3' \) for some threaded \( T_1' \), \( T_3' \) and \( \Gamma \vdash_D P \Rightarrow \Delta' \cdot s : T_1' \cdot T_3' \) for some \( \Delta' < \Delta \) and threaded \( T_1' \) such that \( \Delta' < \Delta \). Moreover, if \( T' = \bot \), \( T_1' = T_1 \).

3. If \( \Gamma \vdash_D s([s]); P \Rightarrow \Delta \cdot s : T_1 \) and \( T_3 \) is not threaded, then \( T_3 = ![T_1]; T_3' \) for some threaded \( T_0 \) and non-threaded \( T_3' \) and \( \Gamma \vdash_D P \Rightarrow \Delta' \cdot s : T_1 \) for some \( \Delta' < \Delta \) and threaded \( T_1 \).

4. If \( \Gamma \vdash_D s([s]); P \Rightarrow \Delta \cdot s : T_1 \) and \( T_3 \) is threaded, then \( T_3 = ![T_1]; T_3' \) for some threaded \( T_1' \) and \( \Gamma \vdash_D P \Rightarrow \Delta' \) for some \( \Delta' < \Delta \).

The same holds for \( \vdash_R \).

**Proof:** In the derivation tree we have the \([\text{Ex-Thr}]\) node s.t. \( \Gamma \vdash_D s([t]); P \Rightarrow \Delta' \cdot s : ![T_0]; T_1 \cdot t : T_1 \) for 1 and 2 and \( \Gamma \vdash_D s([s]); P \Rightarrow \Delta' \cdot s : ![T_0]; T_1 \cdot t : T_1 \) for 3 and 4 respectively, for some \( T_0, T_1' \) and \( \Delta' < \Delta \). Here \( T_1' \) is threaded for 1,2 and 4 and not threaded for \( T_1' \).

(1) From premise of \([\text{Ex-Thr}]\), \( \Gamma \vdash_D P \Rightarrow \Delta_0 \) and \( \Delta_0 \circ_D t : T_0 = \Delta' \cdot s : T_1 \cdot t : T_1 \) hold for some \( \Delta_0, T_0 \). Since \( T_1 \) is neither \( T \) nor \( \text{Odd} \) we obtain \( t \notin \text{dom}(\Delta_0) \), \( T_1 = T_0 \) and \( \Delta_0 = \Delta' \cdot s : T_1 \). Immediately from that, \( \Gamma \vdash_D P \Rightarrow \Delta' \cdot s : T_1' \) and \( T_3 = ![T_1]; T_3' \) hold. The same holds for \( \vdash_R \), since by its definition \( \text{Odd} \) \( \neq \text{cod}(\Delta') \) and \( T_1 \neq \text{Odd} \).

(2) Let \( T_0 = T_1 \) for some \( T_1 \). From premise of \([\text{Ex-Thr}]\), \( \Gamma \vdash_D P \Rightarrow \Delta_0, \Delta_0 \circ_D t : T_1 \) and \( \Delta_0 \circ_D t : T_1 = \Delta' \cdot s : T_1' \cdot t : T_1' \) hold for some \( \Delta_0 \). Since \( T' \) is not threaded, \( t \in \text{dom}(\Delta_0) \) and \( \Delta_0 = \Delta' \cdot s : T_1' \cdot t : T_1' \) hold for some \( T_1' \), and both
$T_i$ and $T'_i$ are threaded. In case $T' = \perp$, $T_i = \overline{T_i}$ and $\Gamma \vdash_D P \triangleright \Delta' \cdot s : T_i \cdot t : \overline{T_i}$ hold. Otherwise $T' = \oddd$ and $\Gamma \vdash_D P \triangleright \Delta' \cdot s : T_i \cdot t : \overline{T_i}$ hold. For $\forall r$, all we need is the former $T' = \perp$ case only.

(3) From premise of $[\text{Ex-Tm}]$, $\Gamma \vdash_D P \triangleright \Delta_0$ and $\Delta_0 \circ_D s$ s $T_0 = \Delta' \cdot s : T'_i$ for some $\Delta_0$. Since $T'_i$ is not threaded, $s \in \text{dom}(\Delta_0)$ and $\Delta_0 = \Delta' \cdot s : T_i$ for some $T_i$, and from $\Delta_0 \circ_D s : T_{00}$, both $T_0$ and $T_1$ are threaded. Immediately from that, $\Gamma \vdash_D P \triangleright \Delta' \cdot s : T_i$ holds. The same holds for $\forall r$, since $\forall r \notin \text{cod}(\Delta')$ and $T_i$ is threaded.

(4) From premise of $[\text{Ex-Tm}]$, $\Gamma \vdash_D P \triangleright \Delta_0$ and $\Delta_0 \circ_D s : T_0 = \Delta' \cdot s : T'_i$ for some $\Delta_0$. Since $T'_i$ is threaded, $T_0 = T'_i$ and $\Delta_0 = \Delta'$ hold. Immediately from that, $\Gamma \vdash_D P \triangleright \Delta'$ holds. For $\forall r$, from assumption $\forall r \notin \text{cod}(\Delta')$ holds.

Types are preserved over structural congruence.

**Lemma 5** (Subtype Congruence): If $\Gamma \vdash_D P \triangleright \Delta$ and $P \equiv Q$, then $\Gamma \vdash_D Q \triangleright \Delta$.

**Proof:** The proof follows that in [9]. The proof of typeability of the subterm uses inversion lemma (Lemma 3).

The following proposition shows that $R$ is a conservative extension of the one in [1].

**Proposition 2:** If $\Gamma \vdash P \triangleright \Delta$, then $\Gamma \vdash_D P \triangleright \Delta$.

**Proof:** By simple induction on the derivation of the typing judgment. See that the replacement of the rules in Fig. 5 does not change typings from $\vdash$.

Then we confirm that even if we include more processes to be typeable in $R$, it actually rejects error terms.

**Lemma 6** (Not an error if typeable): If $\Gamma \vdash_D P \triangleright \Delta$, then $P$ is not an error.

**Proof:** Assume $P$ is an error, namely $P \equiv (v3)((Q_1|Q_2)|R)$ where $Q_1$ and $Q_2$ are s-processes but $Q_1|Q_2$ is not an s-redex. By Lemma 5, there exist $\Delta'$ such that $\Gamma \vdash (v3)((Q_1|Q_2)|R) \triangleright \Delta'$. By Inversion lemma (Lemma 3), there exist $\Delta_1$ and $\Delta_2$ such that $\Gamma \vdash Q_1|Q_2 \triangleright \Delta_1 \circ_D \Delta_2$ and for $i \in [1, 2], \Gamma \vdash Q_i \triangleright \Delta_i$ holds. Since each $Q_i$ is an s-processes but $Q_1|Q_2$ is not an s-redex, $(\Delta_1 \circ_D \Delta_2)(s) = \oddd$, hence $\forall r \notin \text{cod}(\Delta')$. This contradicts $\Gamma \vdash_D P \triangleright \Delta$.

5.2 Subject Reduction and Type Safety

We proceed to the proofs of subject reduction and type-safety, which is one of the main results in the current paper. Subtle part of the proof lies in the substitution. We prove three lemmas for preservation of subjectability over substitutions on channels.

The substitution of channels only occurs via delegation $s[\langle i \rangle | P] \cdot s'[\langle u \rangle | Q] \cdot s''[\langle v \rangle | R] \longrightarrow P | Q | R$ (Pass rule in the Fig. 2), hence we only consider two cases, (i) $t \notin \text{fc}(Q)$ or (ii) the type of $t$ in $Q$ is dual to that of $u$, in other words $\forall r \vdash Q \triangleright \Delta \cdot t : \Gamma$ for some $\Delta, T$.

**Lemma 7** (First substitution lemma): The following holds:

1. $\Gamma \vdash_{D} P \triangleright \Delta$ and $s \notin \text{dom}(\Delta)$, then $\Gamma \vdash_{D} P[t/s] \triangleright \Delta$.
2. $\Gamma \vdash_{D} P \triangleright \Delta \cdot s : T_i$ and $t \notin \text{dom}(\Delta)$, then $\Gamma \vdash_{D} P[t/s] \triangleright \Delta \cdot t : T_i$.

The same also holds for $\forall r$.

**Proof:** The first one is trivial. For the second one, proof goes by induction on the height of the derivation tree of $\forall r$.

For the case (ii), we cannot use a simple induction on the derivation tree. We want to try to prove that $\forall r \vdash \Delta \cdot s : T \cdot t : \overline{T}$ implies $\forall r \vdash P[s/t] \triangleright \Delta'$ for some $\Delta'$, and $\forall s.\forall \Delta(s') \neq \oddd$. However, in a case that $P$ is prefixed by $s$, we cannot use induction hypothesis. For example, see that in $\forall r \vdash s[\langle e \rangle | P] \triangleright \Delta \cdot s : T \cdot t : \overline{T}$ the subterm $\Gamma \vdash P'[\langle e \rangle | P] \triangleright \Delta \cdot t : ?s;T \cdot t : ?s;\overline{T}$ does not match with induction hypothesis, since type of $s$ and $t$ is not dual i.e. $\overline{T} 
\neq ?s;\overline{T}$. For such cases we prepare substitution lemma for $D$ that allows typng to become unsafe by such substitution.

**Lemma 8** (Substitution lemma for $D$): If $\Gamma \vdash_D P \triangleright \Delta \cdot s : T_i \cdot t : T_i$ and $T_i, T_i$ are threaded, then there exists $T'$ such that $\Gamma \vdash_D P[s/t] \triangleright \Delta \cdot s : T'$.

**Proof:** The proof proceeds by structural induction on a derivation tree of the type judgment. Notice that $T'$ can be $\oddd$. We only consider cases where $P$ is a parallel composition or a prefixed process. The other cases are trivial.

If $P = P_1|P_2$, there are two cases to consider. Both uses inversion lemma. (1) Case $\Gamma \vdash P_i \triangleright \Delta_i \cdot s : T_i \cdot t : T_i$ for either $i = 1$ or $i = 2$. Then, we can proceed by using induction hypothesis. (2) Case $\Gamma \vdash P_1 \triangleright \Delta_1 \cdot s : T_i$, $\Gamma \vdash P_2 \triangleright \Delta_2 \cdot t : T_i$, $\Gamma \vdash P_2[s/t] \triangleright \Delta_2 \cdot s : T_i$. Since $T_i$ and $T_i$ is threaded, its dual is also defined and we can obtain $\Gamma \vdash (P_1|P_2)[s/t] \triangleright \Delta \cdot t : T'$ and $T' = \perp$ or $T' = \oddd$ depending on $T_i = \overline{T_i}$ or not.

For prefixed processes, we can easily obtain $T'$ by just using induction hypothesis in the premise of the corresponding typing rule, and applying that rule again. However, a bit subtle case lies in $[\text{Ex-Tm}]$, where $P = u!\langle u' \rangle | P'$ for some $u, u', P'$. Here we only consider the case $u, u' \in [s, t]$, since other cases are trivial.

(T1) Case $P = s[\langle i \rangle | P']$. From 1. of Lemma 4, we get

$\Gamma \vdash_D P' \triangleright \Delta \cdot s : T_i$ for some $T_i$ such that $T_i = [T_i]; T_i'$. By Lemma 7 we get $\Gamma \vdash_D P'[s/t] \triangleright \Delta \cdot s : T_i'$, and we obtain $\Gamma \vdash_D s[\langle i \rangle | P'] \triangleright \Delta \cdot s : [T_i]; T_i$ for some $T_i, T_i$ where $s : T_i \circ_D s : T_i = s : T_i$, by applying $[\text{Ex-Tm}]$. (T2) Case $P = s[\langle i \rangle | P']$ is similar to (T1).

(T3) Case $P = s[\langle i \rangle | P']$. From 4. of Lemma 4, we get

$\Gamma \vdash_D P' \triangleright \Delta \cdot t : T_i$. By Lemma 7 we get $\Gamma \vdash_D P'[s/t] \triangleright \Delta \cdot s : T_i$. By applying $[\text{Ex-Tm}]$, we obtain $\Gamma \vdash_D P'[s/t] \triangleright \Delta \cdot s : [T_i]; T_i$ for some $T_i, T_i$ where $T_i \circ_D T_i = T_i'$. (T4) Case $P = s[\langle i \rangle | P']$ is similar to (T3).

Lemma 8 says that, if we have $T_i \neq \overline{T_i}$, then $P[s/t]$ can be an error. However, safe communication involve only the case $T_i = \overline{T_i}$, and in such case we can ensure that $P$ is not
an error, as we will show in the next lemma.

By using Lemma 8, we prove the following essential substitution lemma for R.

**Lemma 9 (Substitution lemma for R):** If \( \Gamma \vdash_R P \Rightarrow \Delta \cdot s : T' \cdot t : T \), then \( \Gamma \vdash_R P[s/t] \Rightarrow \Delta \cdot s : T' \) for some \( T' \).

**Proof:** Simple induction on a derivation tree of the type judgment. The important case is that in which \( P \) is prefixed by \( s \) or \( t \), where Lemma 8 is used instead of induction hypothesis. Let us sketch the proof by considering the case \( P = s!(\text{true}); P' \) for some \( P' \). It follows that \( \Gamma \vdash_R P' \Rightarrow \Delta \cdot s : T' \cdot t : T \) where \( \Delta \cdot s \in \text{fc}(P) \) is included and that is necessary to establish the proof of subject reduction. As a corollary we obtain the following extension of the substitution lemma for R.

**Corollary 1 (Substitution lemma for \( \vdash_P \)):** If \( \Gamma \vdash P \Rightarrow \Delta \cdot s : T' \cdot t : T \cdot s/\text{true}; P' \) is not error.

**Proof:** Immediately holds from Lemma 6 and Lemma 9.

Accordingly, some substitution introduces \( \text{Odd} \) type which indicates an error, but it is only occurs under a prefix. Now we state that the \( \text{Odd} \) is never exposed i.e. a process never reaches the error state. Before that, we prepare the following lemma that states the type of a channel where the communication occurs.

**Lemma 10 (Type of s-redex):** If \( \Gamma \vdash_R P \Rightarrow \Delta \) and \( P \) is a s-redex, then \( \Delta(s) = \bot \).

**Proof:** If that \( \Delta(s) \) is prefixed, by a straightforward induction on the derivation tree of the type judgment \( P \) is not a s-redex, which contradicts with the assumption. Since \( \forall t, \Delta(t) \neq \text{Odd} \), \( \Delta(s) = \bot \).

Then, the main theorem of the current paper, subject reduction and type safety of R, is established by the next theorem.

**Theorem 1 (Subject reduction):** For all \( t \), if \( \Gamma \vdash_R P \Rightarrow \Delta \cdot t : T \) and \( P \rightarrow P' \) then \( \Gamma \vdash_R P' \Rightarrow \Delta \cdot t : T'_t \). Moreover, if \( T_t \) is prefixed, \( T'_t = T'_s \).

**Proof:** We proceed by induction on derivation tree of \( P \rightarrow P' \). We only consider the simple case \( s!(\text{true}); P_1 \parallel s?((u); P_2 \rightarrow P_1 \parallel P_2[t/u] \) which involves substitution and type change. Other cases does not involve type change, hence \( T_t = T'_s \).

(25) Follows that \( T_t = \bot \). By Inversion lemma (Lemma 3 and 1. of Lemma 4) and premise of the typng rule \([\text{Car}]\), the derivation tree of the type judgment has the following node:

\[
\begin{align*}
\Gamma &\vdash_R P \Rightarrow \Delta'_s \cdot s : T_s \\
\Gamma &\vdash_R s!(\text{true}); P_1 \Rightarrow \Delta'_s \cdot s : \text{!}[T_1];T_s \cdot t : T_1
\end{align*}
\]

and

\[
\begin{align*}
\Gamma &\vdash_R P_2 \Rightarrow \Delta'_s \cdot s : T_s \\
\Gamma &\vdash_R s?(u); P_2 \Rightarrow \Delta'_s \cdot s : ?[T_1]; T_s \cdot t : T_1
\end{align*}
\]

where \( \Delta'_s \cdot s = \Delta' \) and \( \Delta'_s < \Delta' \). By Substitution lemma for R (Lemma 9), we obtain \( \Gamma \vdash_R P_2[t/u] \Rightarrow \Delta'_s \cdot s : T_s \cdot t : T' \) for some \( T' \). Applying [Conc], we get that \( \Gamma \vdash_R P_1 \parallel P_2[t/u] \Rightarrow \Delta' \cdot \bot 

\text{communication occurs.} \]

Finally, the type safety is easily obtained from Lemma 1 and Theorem 1.

**Theorem 2 (Type safety):** If there exists \( \Delta \) such that \( \vdash_R P \Rightarrow \Delta \), then \( P \) does not reduce to an error.

**Proof:** Immediately holds from Lemma 6 and Theorem 1.

As a corollary, we show that the original session type system also [1] holds communication safety.

**Corollary 2 (Type safety of [1]):** If \( \vdash P \), then for all \( P' \) s.t. \( P \rightarrow P' \), \( P' \) is not an error.

**Proof:** Immediately holds from Proposition 2 and Theorem 1.

6. Related Work

Yoshida and Vasconcelos proposed a session type system with polarity in [9]. Their session-type system of polarized process calculus distinguishes the two ends of a channel syntactically. A ‘forwarder’ process

\[
fwd!(t); ?(x : \text{int}); s!((x); 0)
\]

may be written with polarity annotation:

\[
fwd^+?((t); ?(x : \text{int}); s^+!(x)); 0
\]

Every channel occurring freely in a process is annotated with polarities such as \( s^+ \) and \( s^- \). In (26), by receiving \( s^- \) at \( fwd^+ \), it evolves to the following:

\[
s^-?(x : \text{int}); s^+!(x); 0
\]

Unlike the one for non-polarized \( \pi \)-calculus, their type system can give distinct types to each end, hence type change problem is avoided.

Giunti et al. [12] proposed another proof of type safety of session types for the \( \pi \)-calculus, by giving a translation from the session-typed \( \pi \)-calculus to their double binder language, which is much similar to the polarized calculus of Yoshida et al. The proof is a bit complex, since it requires both type-safety of the target language and soundness of the translation to hold.

Giunti et al. [13] proposed another session type system for the \( \pi \)-calculus in terms of linear types, with type preservation. It tracks the two parallel use of channels as the pair
of session types \((T, T')\), while the our type system fills it as \(\perp\) or Odd. By two dedicated typing rules that prefixes on such pair type, it preserves a session type balanced during reduction, even for cases that our type system changes the type \(\perp\) to some prefixed type. It is explained in our notation as follows. For example, the process (15) we have seen, \(s!(\text{true}); s?!(x); 0 | s?(y); s?(y); 0\) is typeable under a pair type \(s : (\!S; 1; \!S; 2; \text{end}; \?S; 1; \!S; 2; \text{end})\), which is obtained by the [T-OutC] rule of [13] that prefixes the left hand of the the pair type \(s : (\?S; 2; \text{end}; \?S; 1; \!S; 2; \text{end})\) of the subterm with \(\!S; 1\).

Our type system has advantages in the following two points: (i) Our type system has one unique typing rule for each process construct, whereas theirs has each two rules for input prefix and output prefix, [T-Ins] and [T-InsC], and [T-Out] and [T-OutC]. It will make simple unification-based type inference be cumbersome, while our type system enabled us to encode session type in Haskell [7] by using the type-inference mechanism of that language. (ii) Our type system clarifies the inconsistency of session after a channel-passing, by marking them as Odd. In their type system such process will appear in the premise of each use of [T-InsC] and [T-OutC] rule.

Also, we have [Ex-ThrK] rule, which makes some useful delegating processes typeable. Though they say their rule also allows \(s!(\langle s \rangle)\); \(P\), it just gives unrestricted (non-linear) type via recursion of types, and that point is different from ours. Our typing rule justifies them for even linear uses of a channel, by treating it as “delegate the rest of the usage by itself” manner in the special case of [Ex-ThrK] rule. Their [T-Out] rule is following:

\[
\begin{align*}
\Gamma_1 \vdash v : T & \quad \Gamma_2, x : S + P \\
\Gamma_1 \cdot (\Gamma_2, x : \text{lin}(T, S) + \bar{x}(y)) \cdot P
\end{align*}
\]

Following the [Ex-ThrK] rule, this can be reformulated as

\[
\begin{align*}
\Gamma_1 \vdash v : T & \quad \Gamma_2 \vdash P & \quad \Gamma_1 \cdot \Gamma_2 = \Gamma_3, x : S \\
\Gamma_3, x : \text{lin}(T, S) + \bar{x}(y) \cdot P
\end{align*}
\]

Then, it will allow the process \(\Gamma, \text{lin}(T, S) + \bar{x}(x), P\) as a special case that \(x = v\) and \(T = S\).

7. Conclusion

We have proposed a session type system \(\mathbf{R}\) with the subject reduction property. The key technique is to allow type assigned to processes change while evolving processes. We have shown the subject reduction that ensures ‘unsafe type’ Odd is never exposed. The proof of the substitution lemma is divided into two parts, because in some cases the prefixed process does not allow the straightforward induction, which is the root of the failure of subject reduction in [1]. By using \(\mathbf{R}\), an alternative proof of the original binary session types for the session-typed \(\pi\)-calculus [1] is presented.

The novel typing rule [Ex-ThrK] in Fig. 5 covers more processes to be typeable in a uniform manner. We have implemented identical typing discipline in our implementation of session types in Haskell [18], which allows program like self-delegation \(s!(\langle s \rangle)\); \(P\) in Sect. 4.2 to be typeable. To the authors’ knowledge, it has not been typeable in the existing session type systems ever. We believe \(\mathbf{R}\) extends the application of session type in a practical sense.

Future work includes construction of new rules that makes more processes to be typeable in this direction. For example, the rule

\[
\begin{align*}
\Gamma \vdash P & \triangleright \Delta \cdot s : \perp \\
\Gamma \vdash s!(e); P & \triangleright \Delta \cdot s : \!S; \text{end}
\end{align*}
\]

would justify the reuse of a channel \(s\) in one endpoint. When the communication have done, (in the above case some value was sent on that channel), the continuation of that channel can use it differently in parallel. The class of such types should be explored by following the generic type system for the \(\pi\)-calculus [19] of Igarashi and Kobayashi.

The relaxed typing rules of \(\mathbf{R}\) sometimes give types to even undesirable, deadlocked terms that are correctly rejected by the original session type system [1]. This is understood as difference of target terms, i.e. ours is run-time property hold during reduction, while the original one is compile-time check before running it. The whole typing discipline in \(\mathbf{R}\) is used to prove safety of all running processes. On the other hand, compile-time check should obey the followings to rule out obvious deadlocks. (i) Any occurrences of Odd must be rejected, since it always appears in a deadlocked term. In other words, \(\circ\) and \(\times\) must be used instead of \(\circ_d\) and \(\times_p\), respectively. (ii) Rules [CR_D] and [ConC] must be used instead of relaxed [Ex-CR_D] and [Ex-ConC], respectively. Even in that setting, \(\mathbf{R}\) enjoys more number of processes to be typeable by [Ex-ThrK]. Such distinction between run-time compile-time can also be seen in stupid cast of Featherweight Java [20]. Stupid cast is included in the type system to retain subject reduction of Java, whereas it is not included in a compiler since it gives types to erroneous cast.

Acknowledgments

The authors thank anonymous reviewers for their helpful comments on an earlier version of this paper. The first author acknowledges and thanks his colleagues in IT Planning Inc. for supporting him during the writing of this paper. This work was partially supported by the Grant-in-Aid (Scientific Research (B) 20300009 and Scientific Research (C) 19500026) from the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

References

Appendix: Rest of the proof of Theorem 1

(Case $s = t$) From Lemma 10, we obtain $\Gamma \vdash_R s!⟨s⟩; P_1 \vdash Δ · s : T$. By Inversion lemma (Lemma 3 and 4. of Lemma 4) and premise of the typing rule $[Cf]$, the derivation tree of the type judgment has the following node:

$$\begin{array}{c}
\Gamma \vdash_R P_1 \vdash Δ'_{1} \\
\Gamma \vdash_R s!(s); P_1 \vdash Δ'_{1} · s : ![T]; T
\end{array} \quad (A-1)$$

and

$$\begin{array}{c}
\Gamma \vdash_R P_2 \vdash Δ'_{2} · s : T; u : T \\
\Gamma \vdash_R s?(u); P_2 \vdash Δ'_{2} · s : ![T]; T
\end{array} \quad (A-2)$$

where $Δ'_{1} ◦ D Δ'_{2} = Δ'$ and $Δ' < Δ$. By Substitution lemma for $R$ (Lemma 9), we obtain $Γ \vdash_R P_2[s/u] \vdash Δ'_{2} · s : T'$ for some $T'$. Applying $[Conc]$, we get that $Γ \vdash_R P_1 | P_2[t/u] \vdash Δ' · s : T'$. In addition, by applying $[Bor]$ we conclude that $Γ \vdash_R P_1 | P_2[t/u] \vdash Δ : s : T'$. 

(Case $s ≠ t$) From Lemma 10, we obtain $Γ \vdash_R s!(t); P_1 | s?(u); P_2 \vdash Δ · s : \bot · t : T_i$ for some $T_i$.

(Sub-case $t \notin fc(P_2)$) If $t \notin fc(P_2)$, it is trivial to show $Γ \vdash_R P_1 | P_2[t/u] \vdash Δ · s : \bot · t : T_i$ by using the First Substitution lemma (Lemma 7). Consider the case $t ∈ fc(P_1)$. By Inversion lemma (Lemma 3 and 2. of Lemma 4) and premise of the typing rule $[Cf]$, the derivation tree of the type judgment has the following node:

$$\begin{array}{c}
\Gamma \vdash_R P_1 \vdash Δ'_{1} · s : T_s · t : T_1 \\
\Gamma \vdash_R s!(s); P_1 \vdash Δ'_{1} · s : ![T_1]; T_s · t : \bot
\end{array} \quad (A-3)$$

and

$$\begin{array}{c}
\Gamma \vdash_R P_2 \vdash Δ'_{2} · s : T_r · u : T_1 \\
\Gamma \vdash_R s?(u); P_2 \vdash Δ'_{2} · s : ![T_1]; T_r
\end{array} \quad (A-4)$$

where $Δ'_{1} ◦ D Δ'_{2} = Δ'$ and $Δ' < Δ$. By First Substitution Lemma (Lemma 7), we obtain $Γ \vdash_R P_2[t/u] \vdash Δ' · s : \bot · t : T_i$. Applying $[Conc]$, we get that $Γ \vdash_R P_1 | P_2[t/u] \vdash Δ' · s : \bot · t : \bot$. In addition, by applying $[Bor]$ we conclude that $Γ \vdash_R P_1 | P_2[t/u] \vdash Δ : s : \bot · t : \bot$. 


Keigo Imai was born in 1982. He received his PhD degree of Information Science from Nagoya University in March, 2012. In 2009, he withdrew from the doctoral program at Nagoya University and became a Researcher in the Center for Embedded Computing Systems, Nagoya University. From April 2010, he is working at IT Planning Inc. as a software engineer. His main research interest is about functional programming, process calculus and concurrency. He is a member of JSSST and ACM.

Shoji Yuen was born in 1963. He graduated Kyoto University in 1985 and received the Doctor of engineering from Nagoya University in 1997. Currently he is a professor in the graduate school of information science of Nagoya University, Japan. His main research interest is about the concurrency theory and practice. He is a member of IPSJ, JSSST, ACM and IEEE.

Kiyoshi Agusa received his PhD degree in computer science from Kyoto University in 1982. Currently, he is a professor of the department of Information Systems, Graduate School of Information Science, Nagoya University. His research area includes software engineering, program repository, and software reuse. He is Vice Director of Information and Communication Headquarters and Director of Information Technology Center, Nagoya University. He is a member of IPSJ, JSSST, ACM and IEEE.