Context-Free Session Types for Applied Pi-Calculus

Jens Aagaard* Hans Hüttel† Mathias Jakobsen ‡ Mikkel Kettunen §
Department of Computer Science, Aalborg University, Aalborg, Denmark

We present a binary session type system using context-free session types to a version of the applied pi-calculus of Abadi et. al. where only base terms, constants and channels can be sent. Session types resemble process terms from BPA and we use a version of bisimulation equivalence to characterize type equivalence. We present a quotiented type system defined on type equivalence classes for which type equivalence is built into the type system. Both type systems satisfy general soundness properties; this is established by an appeal to a generic session type system for psi-calculi.

1 Introduction

Binary session types [6] describe the protocol followed by the two ends of a communication medium, in which messages are passed. A sound type system of this kind guarantees that a well-typed process does not exhibit communication errors at runtime. Session types have traditionally been used to describe linear interaction between partners [10], but later type systems are able to distinguish between linear and unlimited channel usages. In particular Vasconcelos has proposed session types with lin/un qualifiers that describe linear interaction as well as shared resources [10]. *Context-free session types* introduced by [9] are more descriptive than the regular types described in previous type systems [6, 10] in that they allow full sequential composition of types. Because of this, session types can now describe protocols that cannot be captured in the regular session types, such as transmitting complex composite data structures.

Many binary session type systems are concerned with languages that use the selection and branching constructs introduced in [6] in addition to the normal input and output constructs. These constructs are synchronisation operations, where two processes synchronise on a channel, and are similar to method invocations found in object-oriented programming. A branching process $l \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$ continues as process P_k together with P if label l_k is selected on channel c using the selection $c \lhd l_k.P$.

In this article we consider a session type system for a version of the applied pi-calculus, due to Abadi et. al. [1]. This is an extension of the pi-calculus [8] with terms and extended processes and in our case extended further with selection and branching. Our version is a "low-level" version in that allows one to build composite terms but only allows for the communication of names and nullary function symbols. In this way, the resulting version is close to the versions of the pi-calculus used for encoding composite terms [8].

Our session type system combine ideas from the type systems from [9] and [10] into a type system in which session types are context-free and use lin/un qualifiers in order to distinguish between linear and unbounded resources. The resulting type system uses types that are essentially process terms from a variant of the BPA process calculus [4].

In Section 2 we present our version of the applied pi-calculus and in Section 3 we define the syntax and semantics of our session type system. We then prove the soundess of our type system by using

*jbaa14@student.aau.dk
†hans@cs.aau.dk
‡msja15@student.aau.dk
§mkettu16@student.aau.dk

J.A. Pérez and S. Tini (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2018).
EPTCS 276, 2018, pp. 3–18, doi:10.4204/EPTCS.276.3

the general results about psi-calculi from [7]. This is done by showing that the applied pi-calculus is a psi-calculus, and that our type system is a instance of the generic type system presented by Hüttel in [7].

Lastly we present type equivalence between types by introducing a notion of type bisimulation for endpoint types. By considering equivalence classes under type bisimilarity we get a new quotiented type system whose types are equivalence classess. It then follows from the theorems in [7] that the general results for our first type system also hold for the quotiented type system.

2 Applied pi-Calculus

We consider a "low-level" version of the *applied pi-calculus* [1] in which composite terms are allowed but only the transmission of simple data is possible: Only names n, constants represented by functions f_0 with arity 0 and functions that evaluate to values of base types can be transmitted. We use the notation $\widetilde{M}$ to represent the sequence $M_1,\ldots,M_i$ and $\widetilde{x}$ to represent the sequence of variables $x_1,\ldots,x_i$. We always assume that our processes are specified relative to a family of parameterized agent definitions that are on the form $N(\widetilde{x}) \stackrel{\text{def}}{=} A$ and that every agent variable N occurring in a process has a corresponding definition.

The formation rules for processes P and extended processes A can be seen below in (1). Note that we distinguish between variables ranged over by $x,y\ldots$ and names ranged over by $m,n,\ldots$. We let a range over the union of these sets. We extend the syntax of processes with branching $c \rhd \{l_1 : P_1,\ldots,l_k : P_k\}$ and selection $c \lhd l.P$ where $l_1,\ldots l_k$ are taken from a set of labels.

Extended processes extend processes with the ability to use *active substitutions* of the form $\{^M/_x\}$ that instantiate variables.

$$\begin{aligned} P &::= \mathbf{0} \mid P_1 \mid P_2 \mid !P \mid (\nu\, n)P \mid (\nu\, n)P \mid \texttt{if } M_1 = M_2 \texttt{ then } P_1 \texttt{ else } P_2 \\ &\quad \mid n(x).P \mid n\langle u\rangle.P \mid c \lhd l.P \mid c \rhd \{l_1 : P_1,\ldots,l_k : P_k\} \mid N(\widetilde{M}) \\ u &::= n \mid x \mid f_0 \\ M &::= n \mid x \mid f(\widetilde{M}) \\ A &::= P \mid A_1 \mid A_2 \mid (\nu n)A \mid (\nu\, x)A \mid \{^M/_x\} \end{aligned} \tag{1}$$

The notion of structural congruence extends that of the usual pi-calculus [8]. The following two further axioms that are particular to the applied pi-calculus are of particular importance, as they show the role played by active substitutions.

$$P \mid \{^M/_x\} \equiv P\{^M/_x\} \mid \{^M/_x\} \qquad\qquad \mathbf{0} \equiv (\nu\, x)\{^M/_x\}$$

Together with the axioms of [8] they allow us to factor out composite terms such that they only occur in active substitutions. For instance, we have that $\texttt{if } M_1 = M_2 \texttt{ then } P \texttt{ else } Q \equiv \nu x \nu y(\texttt{if } x = y \texttt{ then } P \texttt{ else } Q \mid \{^{M_1}/_x\} \mid \{^{M_2}/_y\})$. We can therefore use a term by only mentioning the variable associated with it. In our version of the applied pi-calculus we very directly make use of this.

Our semantics consists of reduction semantics and a labelled operational semantics that extend those of [1] with rules for branching and selection. Reductions are of the form $P \to P'$, while transitions are of the form $P \xrightarrow{\alpha} P'$ where the label α is given by

$$\alpha ::= c \lhd l \mid c \rhd l \mid a(x) \mid \overline{a}x \mid \tau$$

and $c \lhd l$ is the co-label of $c \rhd l$. The rules defining reductions and labelled transitions are found in Tables 1 and 2, respectively.

(COM)	$x\langle y\rangle.P \mid x(y).Q \to P \mid Q$
(SELECT)	$c \lhd l_k.P \mid c \rhd \{l_1 : P_1, \ldots, l_n : P_n\} \to P \mid P_k \quad \text{for } 1 \leq k \leq n$
(MATCH-TRUE)	$\texttt{if } M = M \texttt{ then } P \texttt{ else } Q \to P$
(MATCH-FALSE)	$\texttt{if } M = N \texttt{ then } P \texttt{ else } Q \to Q$

Table 1: Reduction rules

(RED)	$\dfrac{P \to P'}{P \xrightarrow{\tau} P'}$	(SELECT)	$c \lhd l.P \xrightarrow{c \lhd l} P$
(BRANCH)	$c \rhd \{l_1 : P_1, \ldots, l_k : P_k\} \xrightarrow{c \rhd l_i} P_i$ for $1 \leq i \leq k$	(PAR)	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$ if $\mathrm{bv}(\alpha) \cap \mathrm{fv}(Q) = \emptyset$
(OUTPUT)	$x\langle y\rangle.P \xrightarrow{x\langle y\rangle} P$	(INPUT)	$x(y).P \xrightarrow{x(y)} P$
(NEW)	$\dfrac{P \xrightarrow{\alpha} P'}{(\nu n)P \xrightarrow{\alpha} (\nu n)P'}$ if $n \notin \mathrm{fn}(\alpha)$	(STRUCT)	$\dfrac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad Q' \equiv P'}{P \xrightarrow{\alpha} P}$

Table 2: Labelled transition rules

3 A context-free session type system

We now present the syntax and semantics of our type system.

3.1 Session types

Session types describe the communication protocol followed by a channel. Consider writing a process that transmits a binary tree where each internal node containsan integer. We want to transmit this tree by sending only base types (i.e. the integers) on a channel. The tree data type can be described with the grammar below.

$$\mathsf{Tree} ::= (\mathsf{Int}, \mathsf{Tree}, \mathsf{Tree}) \mid \mathsf{Leaf}$$

Using the regular types introduced in [10], the type for a channel transmitting such a data structure could be the recursive type $\mu z. \oplus\{\mathsf{Leaf} : \mathsf{lin\ skip} \quad \mathsf{Node} : \mathsf{lin\ !Int}.z\}$. In this type z is a type variable that is recursively defined. The type describes how a single node is transmitted, if it is a leaf node we do not do

anything, if it is an internal node then the integer value is transmitted with the output type !Int and then the sub-trees of the node are transmitted with a recursive call.

However, if we use this regular session type, we are not able to guarantee that the tree structure is preserved. The reason is that the session type describes that a list of nodes are being sent, but not the position in the tree of each node. On the other hand, if we use the context-free session type disciple introduced by Thiemann and Vasconcelos [9], we can specify the preservation of tree structures by using types such as $\mu z. \oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; z; z\}$. Using sequential composition with the $\_;\_$ operator, we can specify a protocol that will guarantee that the tree structure by first sending the left sub-tree and then the right sub-tree. Introducing a sequential operator can introduce challenges for typing a calculus, as the following example shows. If we were to reuse a channel by sending an integer after transmitting a tree, the type would be $\mu z. \oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; z; z\}; !\mathsf{Int}$.

$$\frac{\Gamma(c) = \oplus\{l_i : T_{E_i}\}_{i \in I}}{\Gamma \vdash \text{select } l_i \text{ on } c} \tag{2}$$

When typing rules are created for a calculus, it is often defined on the structure of types and terms. An example of such a typing rule for a select statement is shown in (2), which says that a select statement is well typed if the select operation is performed on a channel with a select type. If however the channel has a type as shown before, the select rule cannot be used, as the channel has the type of a sequential composition, and inside the sequential composition we have a recursive type.

We require an equirecursive treatment of types, which allows us to expand the type to $\oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; \mu z. \oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; z; z\}; \mu z. \oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; z; z\}\}; !\mathsf{Int}$. This is achieved by unfolding a recursive type $\mu z.T$ to T where all occurrences of z in T are replaced with the original $\mu z.T$. So now we are left with a sequential composition with a select type and output type. In order to transform this into a select type, we need a distributive law that allows us to move the sequential composition inside the select type, in order to obtain the type $\oplus\{\mathsf{Leaf} : \mathsf{skip}; !\mathsf{Int} \quad \mathsf{Node} : !\mathsf{Int}; \mu z. \oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; z; z\}; \mu z. \oplus\{\mathsf{Leaf} : \mathsf{skip} \quad \mathsf{Node} : !\mathsf{Int}; z; z\}; !\mathsf{Int}\}$. Such a rule does exist, we will introduce a method to prove that these types exhibit the same behaviour in Section 6 when we introduce type equivalence.

3.2 The language of types

We denote the set of all types by $\mathcal{T}$; the types $T \in \mathcal{T}$ are described by the formation rules in (3). These session types are a modified version of the context-free session types presented by [9]. The modifications made to the types are that we allow input and output session types to transmit other session types to allow sending channels in other channels, and finally that we introduce the lin and un qualifiers. We let B range over a set of base types.

$$\begin{aligned} p &::= \mathsf{skip} \mid ?T \mid !T \mid \&\{l_i : T_{E_i}\} \mid \oplus\{l_i : T_{E_i}\} \\ q &::= \mathsf{lin} \mid \mathsf{un} \\ T_E &::= q\ p \mid z \mid \mu z.T_E \mid T_{E_1}; T_{E_2} \\ T &::= S \mid B \mid T_E \\ S &::= (T_{E_1}, T_{E_2}) \end{aligned} \tag{3}$$

From the formation rules we can see that a session type S is a pair of endpoint types T_{E_1} and T_{E_2}. Endpoint types describe one end of channel, and are the types that evolve when a channel is used. The qualifiers lin and un from [10] are used to describe a linear interaction between two partners and a

unrestricted shared resources respectively. An example of an un type could be a server, that a lot of processes have access to. To ensure that no communication errors occur if multiple processes can read from a channel concurrently, we require that the type must never change behaviour. This means that an un type must be the same before and after a transition.

3.3 Transitions for types

We now present the transition rules for endpoint types, which describe how the types can evolve when an action is performed. We use an annotated reduction semantics to describe the behaviour of our types. Our labels are generated by the grammar in (4) where we have select, branch, input and output actions. The transitions are of the form $T_E \xrightarrow{\lambda} T'_E$ and are shown in Table 3. We let $T_E\{y/x\}$ be the endpoint type T_E where all free occurrences of x has been replaced with y.

$$\lambda ::= !T_1 \mid ?T_1 \mid \triangleright l \mid \triangleleft l \tag{4}$$

In the transition rules, we use the function Q defined in (5) to find the qualifier of compound types such as recursive types or sequential composition types.

$$\begin{aligned} Q(q\ p) &\stackrel{\text{def}}{=} q \\ Q(T_{E_1}; T_{E_2}) &\stackrel{\text{def}}{=} Q(T_{E_1}) \\ Q(\mu z.T_E) &\stackrel{\text{def}}{=} Q(T_E) \end{aligned} \tag{5}$$

A relation $\sqsubseteq = \{(\mathsf{lin},\mathsf{un}),(\mathsf{un},\mathsf{un}),(\mathsf{lin},\mathsf{lin})\}$ is defined for qualifiers in [10]. We also follow the definition of $q(T)$ and $q(\Gamma)$ from that of [10]. In short $\mathsf{lin}(\Gamma)$ is always satisfied, and $\mathsf{un}(\Gamma)$ is satisfied iff all elements in Γ are unrestricted.

The type system contains the sequential operator $\_;\_$ as well as choice operators; select $\&\{\ldots\}$ and branch $\oplus\{\ldots\}$. This is very similar to Basic Process Algebra (BPA)[4] that contains the sequential operator $\cdot$ and nondeterministic choice operator $+$. We also have recursive types, which corresponds to variables with recursive definitions in BPA. A BPA expression is *guarded* if all recursive variables on the right hand side is preceded by an action λ [4, p. 53]. Similarly we say that a type is guarded if every recursion variable is preceded by an input or output. These similarities with BPA will become very important, as we can describe types as BPA expressions and by showing results about these expression, we can in turn show results about our types.

3.4 Typing rules

We now present a type system for our version of the applied pi-calculus in Table 8. In this type system, the type judgements for processes are on the form $\Gamma \vdash P$ meaning that the process P is well typed in context Γ. The judgments for terms are on the form $\Gamma \vdash M : T$ meaning that the term M is well typed with type T in context Γ. Lastly the judgments for extended processes are on the form $\Gamma \vdash_A A$ meaning that the process A is well typed in context Γ. We type processes in a type context Γ which contains types for the variables, names and functions symbols of a process. We follow the definition of a type context from [10]: $\emptyset$ is the empty context, $\Gamma, x : T$ is the context equal to Γ except that x has the type T in the new context. This operation is only defined when $x \notin \mathrm{dom}(\Gamma)$.

Table 8 shows the typing rules for processes. We use the context split $\circ$ and context update $+$ operations from [10].

$$\text{(INPUT)}\quad \frac{}{q\ ?T \xrightarrow{?T} q\ \mathsf{skip}} \qquad \text{(OUTPUT)}\quad \frac{}{q\ !T \xrightarrow{!T} q\ \mathsf{skip}}$$

$$\text{(SEQ1)}\quad \frac{T_{E_1} \xrightarrow{\lambda} T'_{E_1} \qquad Q(T_{E_1}) \sqsubseteq Q(T'_{E_1})}{T_{E_1};T_{E_2} \xrightarrow{\lambda} T'_{E_1};T_{E_2}} \qquad \text{(SEQ2)}\quad \frac{T_{E_1} \not\rightarrow \qquad Q(T_{E_1}) \sqsubseteq Q(T_{E_2}) \qquad T_{E_2} \xrightarrow{\lambda} T'_{E_2} \qquad Q(T_{E_2}) \sqsubseteq Q(T'_{E_2})}{T_{E_1};T_{E_2} \xrightarrow{\lambda} T'_{E_2}}$$

$$\text{(SELECT)}\quad \frac{q \sqsubseteq Q(T_{E_k})}{q \oplus \{l_i : T_{E_i}\}_{i\in I} \xrightarrow{\triangleleft l_k} T_{E_k}} \qquad \text{(BRANCH)}\quad \frac{q \sqsubseteq Q(T_{E_k})}{q \,\&\{l_i : T_{E_i}\}_{i\in I} \xrightarrow{\triangleright l_k} T_{E_k}}$$

$$\text{(REC)}\quad \frac{T_E\{\mu z.T_E/z\} \xrightarrow{\lambda} T'_E}{\mu z.T_E \xrightarrow{\lambda} T'_{E_1}}$$

Table 3: Annotated reduction semantics for types

The context split operation is used to split a context into two constituents. A maximum of two processes must have access to a given linear session type; a context either contains the entire session type $S = (T_{E_1}, T_{E_2})$, or a single endpoint type T_E. When splitting a context into two, we can pass S to either context, or one endpoint to each context. If the context only has an endpoint type, the endpoint type can only be passed on to one of the two contexts. This way we ensure that each lin endpoint of a channel is known in exactly one context. Names of unrestricted type can be shared among all contexts.

The context update operation updates the type of a channel. The $\Gamma, x : T$ operation is only defined when $x \notin \mathrm{dom}(\Gamma)$. The $+$ operation $\Gamma = \Gamma_1 + \Gamma_2$ uses the type of x in Γ_2 to update the type in Γ_1. So if $\Gamma_1(x) = T_1$ and $\Gamma_2(x) = T_2$ then $\Gamma(x) = T_2$. The two operations are used in (INPUT) and (OUTPUT) where we must split our context into two, to type each endpoint of a linear channel in its own context.

$$\text{(PLAIN)}\quad \frac{\Gamma \vdash P}{\Gamma \vdash_A P}$$

$$\text{(PAR)}\quad \frac{\Gamma_1 \vdash_A A_1 \qquad \Gamma_2 \vdash_A A_2}{\Gamma_1 \circ \Gamma_2 \vdash_A A_1 \mid A_2}$$

$$\text{(NAME-RES)}\quad \frac{\Gamma, n : S \vdash_A A}{\Gamma \vdash_A (\nu n)A}$$

$$\text{(VAR-RES)}\quad \frac{\Gamma, x : T \vdash_A A}{\Gamma \vdash_A (\nu x)A}$$

$$\text{(SUB)}\quad \frac{\Gamma \vdash x : T \qquad \Gamma \vdash M : T}{\Gamma \vdash_A \{M/x\}}$$

Table 4: Typing rules for Extended Processes

$$\frac{}{\emptyset = \emptyset \circ \emptyset}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad Q(T_E) = \mathsf{un}}{\Gamma, n : T_E = (\Gamma_1, n : T_E) \circ (\Gamma_2, n : T_E)}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad S = (T_{E_1}, T_{E_2}) \quad Q(T_{E_1}) = \mathsf{un} \quad Q(T_{E_2}) = \mathsf{un}}{\Gamma, n : S = (\Gamma_1, n : S) \circ (\Gamma_2, n : S)}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad Q(T_E) = \mathsf{lin}}{\Gamma, n : T_E = (\Gamma_1, n : T_E) \circ \Gamma_2}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad Q(T_E) = \mathsf{lin}}{\Gamma, n : T_E = \Gamma_1 \circ (\Gamma_2, n : T_E)}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad S = (T_{E_1}, T_{E_2}) \quad Q(T_{E_1}) = \mathsf{lin} \quad Q(T_{E_2}) = \mathsf{lin}}{\Gamma, n : S = (\Gamma_1, n : S) \circ \Gamma_2}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad S = (T_{E_1}, T_{E_2}) \quad Q(T_{E_1}) = \mathsf{lin} \quad Q(T_{E_2}) = \mathsf{lin}}{\Gamma, n : S = \Gamma_1 \circ (\Gamma_2, n : S)}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad S = (T_{E_1}, T_{E_2}) \quad Q(T_{E_1}) = \mathsf{lin} \quad Q(T_{E_2}) = \mathsf{lin}}{\Gamma, n : S = (\Gamma_1, n : T_{E_1}) \circ (\Gamma_2, n : T_{E_2})}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad S = (T_{E_1}, T_{E_2}) \quad Q(T_{E_1}) = \mathsf{lin} \quad Q(T_{E_2}) = \mathsf{lin}}{\Gamma, n : S = (\Gamma_1, n : T_{E_2}) \circ (\Gamma_2, n : T_{E_1})}$$

Table 5: Context split for Applied π-calculus, based on [10]

3.5 Duality of types

The notion of type duality is central to session type systems; it expresses that the protocols followed by the two endpoints of a name must be opposites: If one end transmits a value, the other end must receive it. We denote the dual of an endpoint type $\overline{T_E}$ and define duality below, following [9].

$$\frac{}{\Gamma = \Gamma + \emptyset} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, n : T = \Gamma_1 + (\Gamma_2, n : T)}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad Q(T_E) = \mathsf{un}}{\Gamma, n : T_E = (\Gamma_1, n : T_E) + (\Gamma_2, n : T_E)}$$

Table 6: Context update for Applied π-calculus from [10]

$$\text{(NAME)}\quad \frac{\mathsf{un}(\Gamma)}{\Gamma, n : T \vdash n : T} \qquad \text{(VARIABLE)}\quad \frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}$$

$$\text{(FUN)}\quad \frac{\Gamma(f) = T_1 \times T_2 \times \cdots \times T_k \to T}{\Gamma \vdash f : T_1 \times T_2 \times \cdots \times T_k \to T}$$

Table 7: Typing rules for terms

$$\overline{q\ \mathsf{skip}} \stackrel{\text{def}}{=} q\ \mathsf{skip} \qquad \overline{q\ \&\{l_1 : T_{E_1}, \ldots, l_k : T_{E_k}\}} \stackrel{\text{def}}{=} q \oplus \{l_1 : \overline{T_{E_1}}, \ldots, l_k : \overline{T_{E_k}}\}$$

$$\overline{q\ ?T} \stackrel{\text{def}}{=} q\ !T \qquad \overline{q \oplus \{l_1 : T_{E_1}, \ldots, l_k : T_{E_k}\}} \stackrel{\text{def}}{=} q\ \&\{l_1 : \overline{T_{E_1}}, \ldots, l_k : \overline{T_{E_k}}\}$$

$$\overline{q\ !T} \stackrel{\text{def}}{=} q\ ?T \qquad \overline{\mu z.T_E} \stackrel{\text{def}}{=} \mu z.\overline{T_E}$$

$$\overline{T_{E_1}; T_{E_2}} \stackrel{\text{def}}{=} \overline{T_{E_1}}; \overline{T_{E_2}} \qquad \overline{z} \stackrel{\text{def}}{=} z$$

A session type $S = (T_{E_1}, T_{E_2})$ is *balanced* iff its endpoint types are dual, that is, if $\overline{T_{E_1}} = T_{E_2}$. A type context Γ is balanced iff every session type in the range of Γ is balanced. We use the notation $\Gamma \vdash_{\text{bal}} P$ to describe that a process P is well typed in a balanced context Γ.

4 Applied pi-calculus as a psi-calculus instance

Psi-calculus is a general framework for process calculi. In this section we show that the applied pi-calculus is an instance of it, and this will then be used in Section 5 to obtain results about our type system.

An instance of the psi-calculus framework contains the seven elements [3] given in Table 9.

$\mathbf{T}$	Data terms
$\mathbf{C}$	Conditions
$\mathbf{A}$	Assertions
$\dot{\leftrightarrow} : \mathbf{T} \times \mathbf{T} \to \mathbf{C}$	Channel Equivalence
$\otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$	Composition
$\mathbf{1} : \mathbf{A}$	Unit
$\vdash \subseteq \mathbf{A} \times \mathbf{C}$	Entailment

Table 9: Elements of psi-calculi

The syntax of an instance of the psi-calculus is described by the formation rules in (7) that generalize those of the pi-calculus. The input and output constructions allows to use arbitrary terms as channels, and the input construction allows for matching on a pattern $(\lambda \tilde{x})N$; the pattern variables in $\tilde{x}$ are bound to the subterms that match. The other syntactic constructs are similar to those of the pi-calculus. The case construct is a generic case of the if-construct; we generalize match conditions $M_1 = M_2$ to allow any

(NIL)	$\dfrac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$
(PAR)	$\dfrac{\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2}$
(REPL)	$\dfrac{\Gamma \vdash P \quad \mathsf{un}(\Gamma)}{\Gamma \vdash !P}$
(RES)	$\dfrac{\Gamma, n : S \vdash P}{\Gamma \vdash (\nu n)P}$
(IF)	$\dfrac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2 \quad \Gamma \vdash M_1 : T \quad \Gamma \vdash M_2 : T}{\Gamma \vdash \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2}$
(INPUT)	$\dfrac{\Gamma_1 \vdash n : T_E \quad T_E \xrightarrow{?T} T'_E \quad \Gamma_2, x : T + n : T'_E \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash n(x).P}$
(OUTPUT)	$\dfrac{\Gamma_1 \vdash n : T_E \quad T_E \xrightarrow{!T} T'_E \quad \Gamma_2 \vdash M : T \quad \Gamma_2 + n : T'_E \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash \overline{n}\langle M \rangle.P}$
(SELECT)	$\dfrac{T_E \xrightarrow{\triangleleft l} T'_E \quad \Gamma, c : T'_E \vdash P}{\Gamma, c : T_E \vdash c \triangleleft l.P}$
(BRANCH)	$\dfrac{T_E \xrightarrow{\triangleright l_i} T'_E \quad \Gamma, c : T'_E \vdash P_i}{\Gamma, c : T_E \vdash c \triangleright \{l_1 : P_1, \ldots, l_k : P_k\}}$ for all $1 \leq i \leq k$
(AGENT)	$\dfrac{\Gamma \vdash_A A \mid \{\widetilde{M}/\widetilde{x}\}}{\Gamma \vdash N(\widetilde{M})}$ where $N(\widetilde{x}) \stackrel{\text{def}}{=} A$

Table 8: Typing rules for processes

$\phi \in \mathbf{C}$ as a condition. Lastly in psi-calculi we have a concept of assertions $\Psi \in \mathbf{A}$. These generalize the notion of active substition found in the applied pi calculus.

$$P := \mathbf{0} \mid \overline{M}N.P \mid \underline{M}(\lambda\widetilde{x})N.P \mid \mathbf{case}\ \phi_1 : P_1 [] \ldots [] \phi_n : P_n \tag{6}$$

$$\mid\ (\nu a)P \mid P|Q \mid !P \mid (\!|\Psi|\!) \mid M \triangleleft l.P \mid M \triangleright \{l_1 : P_1, \ldots, l_k : P_k\} \tag{7}$$

The structural operational semantics of psi-calculi has transitions of the form $\Psi \blacktriangleright P \xrightarrow{\alpha} P'$ where the labels α are given by the formation rules below [7].

$$\alpha ::= M \triangleleft l \mid M \triangleright l \mid \overline{M}(\nu\widetilde{a})N \mid \underline{K}N \mid (\nu\widetilde{a})\tau@(\nu\widetilde{b})(\overline{M}N\underline{K}) \mid (\nu\widetilde{a})\tau@(M \triangleleft l \triangleright N) \tag{8}$$

The first four actions correspond to the actions we know from the applied pi-calculus: selection, branch, output and input. The last two action are internal τ-actions that correspond to internal actions that are either an input/output-exchange or a branch/select-exchange.

It follows from[3, p. 8] that the standard pi-calculus is an instance of the psi-calculus. Below we do the same for the instance **APi** that shows that the applied pi-calculus is also an instance of the psi-

calculus. In the definition $\mathcal{N}$ is the set of all names and $\mathcal{F}$ is the set of all function names.

$$\mathbf{T} \stackrel{\text{def}}{=} \mathcal{N} \cup \{f(M_1,\ldots M_k) \mid f \in \mathcal{F}, M_i \in \mathbf{T}\} \qquad \mathbf{C} \stackrel{\text{def}}{=} \{M = N \mid M,N \in \mathbf{T}\}$$
$$\mathbf{A} \stackrel{\text{def}}{=} \{1\} \qquad \leftrightarrow \stackrel{\text{def}}{=} \{((n,n), n = n) \mid n \in \mathcal{N}\}$$
$$\otimes \stackrel{\text{def}}{=} \{((\Psi_1,\Psi_2),1) \mid \Psi \in \mathbf{A}\} \qquad \mathbf{1} \stackrel{\text{def}}{=} 1$$
$$\vdash \stackrel{\text{def}}{=} \{(1, M = M) \mid M \in \mathbf{T}\}$$

5 Properties of our type system

A generic binary session type system for psi-calculi was presented in [7]. The intention is that any existing binary session type systems for process calculi can be captured as special instances of the generic system, as long it satisfies four specific requirements. We already know that our applied pi-calculus is a simple psi-calculus and now establish that our type system fulfils the requirements of [7]. This will then allow us to obtain the usual results for binary type systems as a simple corollary of the theorems for the generic system.

5.1 Transition structure

Type transitions in our type system are an instance of the generic type transitions in [7]. Both the generic type transitions and our type transition consist of send, receive, branch and select. The syntax of type transitions is uniform across the two articles as they are both generated by the grammar in (9). This illustrates that there is a one-to-one correspondence between the type transitions in the two type systems.

$$\lambda ::= \triangleleft l \mid \triangleright l \mid !T \mid ?T \tag{9}$$

5.2 Revisiting duality

Duality of types in our pi-calculus is defined on the structure of types, as seen in Section 3.5. In [7] duality is defined on type transitions, where (10) holds for dual types (we use $\underline{\cdot}$ to denote duality defined on type transitions).

$$T_E \xrightarrow{\lambda} T'_E \Leftrightarrow \underline{T_E} \xrightarrow{\overline{\lambda}} \underline{T'_E} \tag{10}$$

In (11) and (12) we see the duality of type transitions, as presented by [7].

$$\overline{!T_1} = ?T_2 \qquad \overline{?T_1} = !T_2 \tag{11}$$
$$\overline{\triangleleft l} = \triangleright l \qquad \overline{\triangleright l} = \triangleleft l \tag{12}$$

We now show that the duality defined in Section 3.5 upholds the property in (10).

Lemma 1. $\underline{T_E} = \overline{T_E}$

Proof. By induction in the structure of types. □

5.3 Checking requirements

In the type system presented in [7], type judgements are relative to a type context and an assignment and therefore of the form $\Gamma, \Psi \vdash \mathscr{J}$, where the judgment body is either $\mathscr{J}$ is either a term typing $M : T$ or P, the statement that process P is well-typed. We write $\Gamma, \Psi \vdash_{\min} \mathscr{J}$, if $\Gamma', \Psi' \not\vdash_{\min} \mathscr{J}$ for every smaller Γ' and Ψ'.

For each requirement presented in [7] we show that it is satisfied in our type system.

Requirement 1: If $\Gamma_1, \Psi_1 \vdash_{\min} \mathscr{J}$ and $\Gamma_2, \Psi_2 \vdash_{\min} \mathscr{J}$ then $\Gamma_1 = \Gamma_2$ and $\Psi_1 \simeq \Psi_2$.

As we only have the assertion $\mathbf{1}$ in type judgements, $\Psi_1 \simeq \Psi_2$ is trivially fulfilled as both are $\mathbf{1}$. Let Γ_1, Γ_2 be type contexts such that $\Gamma_1, \mathbf{1} \vdash_{\min} \mathscr{J}$ and $\Gamma_2, \mathbf{1} \vdash_{\min} \mathscr{J}$. Assume that $\Gamma_1 \neq \Gamma_2$ then without loss of generality there exists an x such that $x \in \mathrm{dom}(\Gamma_1)$ and $x \notin \mathrm{dom}(\Gamma_2)$. Because judgments must be well-formed we know that $fn(\mathscr{J}) \subseteq \mathrm{dom}(\Gamma_1)$ and $fn(\mathscr{J}) \subseteq \mathrm{dom}(\Gamma_2)$, hence $x \notin fn(\mathscr{J})$. Let Γ_1' be defined as Γ_1 except $x \notin \mathrm{dom}(\Gamma_1')$, then $\Gamma_1', \mathbf{1} \vdash \mathscr{J}$ and $\Gamma_1' < \Gamma_1$ thus $\Gamma_1, \mathbf{1} \not\vdash_{\min} \mathscr{J}$. This is a contradiction, hence our assumption is wrong and $\Gamma_1 = \Gamma_2$, which means that the requirement is fulfilled.

Requirement 2: If $\Gamma, \Psi \vdash M : T @ c$ then $\Gamma(c) = T_E$ for some endpoint type T_E.

In our calculus, the only terms that can be used as channels are names. The $(\nu n)P$ construct uses a channel constructor n to declare a channel with a session type $\Gamma, \Psi \vdash n : S@n$. S is a session type $S = (T_{E_1}, T_{E_2})$ for some endpoint types. When type checking n, only one endpoint is present in the context Γ, in which case $\Gamma(n) = T_{E_1}$ or $\Gamma(n) = T_{E_2}$, meaning that the requirement is satisfied.

Requirement 3: Suppose $\widetilde{N} \in \mathrm{MATCH}(M, \widetilde{x}, X)$, $\Gamma, \Psi \vdash M$ and $\Gamma_1 + \widetilde{x} : \widetilde{T}, \Psi_1 \vdash_{\min} X : \widetilde{T} \to U$. Then there exist Γ_{2i}, Ψ_{2i} such that $\Gamma_{2i}, \Psi_{2i} \vdash_{\min} N_i : T_i$ for all $1 \leq i \leq |\widetilde{x}| = n$.

In our calculus, the only possible match is $M \in MATCH(M, x, x)$. As $|\widetilde{x}| = 1$ the requirement becomes $\Gamma_2, \Psi_2 \vdash_{\min} M : T$. From the requirement that M is well typed, this is trivially fulfilled.

Requirement 4: If $\Psi \models M \dot{\leftrightarrow} K$ and $\Gamma, \Psi \vdash M : S$ then $\Gamma, \Psi \vdash K : S$. If $\Psi \models M \dot{\leftrightarrow} K$ and $\Gamma, \Psi \vdash M : T$ then $\Gamma, \Psi \vdash K : \overline{T}$.

In Section (4) we defined $\dot{\leftrightarrow}$ as $=$, meaning that two channels are equal if it is the same name. The first part of the requirement becomes: If $\Psi \models n = n$ and $\Gamma, \Psi \vdash n : S$ then $\Gamma, \Psi \vdash n : S$ which is trivially fulfilled. The second part happens when only one end of a channel is in the context, with an endpoint type, then the other end of the channel must have a dual endpoint type. If $\Psi \models n = c$ and $\Gamma, \Psi \vdash n : T_E$ then $\Gamma, \Psi \vdash c : \overline{T_E}$.

A channel has a balanced session type $S = (T_E, \overline{T_E})$ for some endpoint type. If n and c have endpoint types and are the same channel, then they must also have types dual of each other. If the types are not unrestricted then they can only evolve into other types dual of each other, due to the requirement that only two processes have access to the channels and that if $T_E \xrightarrow{\lambda} T_E' \iff \underline{T_E} \xrightarrow{\overline{\lambda}} \underline{T_E'}$ we know that the types will stay dual of each other. If the types are unrestricted then the requirement that whenever $T_E \xrightarrow{\lambda} T_E'$ for some λ then $T_E = T_E'$, ensures that the types will stay dual of each other.

5.4 Fidelity result

In this section we discuss the results that we obtain by showing that our type system is an instance of the generic type system. Hüttel presents and proves two main theorems for the generic type system, that we will use to show results about our type system as well.

The first theorem, which is about well typed τ-actions from [7] follows below:

Theorem 1 (Well-typed τ-actions, Theorem 9 of [7])**.** Suppose we have $\Psi_0 \blacktriangleright P \xrightarrow{\alpha} P'$, where α is a τ-action and that $\Gamma, \Psi \vdash_{\mathsf{bal}} P$ and $\Psi \leq \Psi_0$ then for some $\Psi' \leq \Psi$ and $\Gamma' \leq \Gamma$ we have $\Gamma', \Psi' \vdash_{\mathsf{min}} \alpha : (T@c, U)$.

This theorem says that if a process P can make a τ-action and become P', and that P is well typed in an environment where channels have pairs of dual endpoint types, then the action is also well-typed. So internal synchronisation in a well-typed process is well typed as well.

The second theorem is about fidelity and follows below:

Theorem 2 (Fidelity, Theorem 10 of [7])**.** Suppose we have $\Psi_0 \blacktriangleright P \xrightarrow{\alpha} P'$, where α is a τ-action and that $\Gamma, \Psi \vdash_{\mathsf{bal}} P$. Then for some $\Gamma' \leq \Gamma$ and for some $\Psi' \leq \Psi$ we have $\Gamma', \Psi' \vdash_{\mathsf{min}} \alpha : (T@c, U)$ and $\Gamma \pm (\alpha, (T@c, U)), \Psi' \vdash_{\mathsf{bal}} \stackrel{\circ}{=} P'$.

This theorem states that when an action performed in a well typed process and the action is well typed, which is guaranteed by the previous theorem, then the resulting process after the τ-action is also well typed in an updated type environment.This result gives us the property that if a process is well typed, then it will never experience communication errors. The theorem also tells us that processes evolve according to the types prescription.

6 Type equivalence

In this section we discuss type equivalence in our type system and show how this leads to a new session type system.

6.1 Why type equivalence matters

First we motivate type equivalence by expressing an example from [9], in our applied pi-calculus and our type system. Recall the example from Section 3.1 where the goal was to transmit a binary tree while preserving the tree structure.

Example 1. Consider the parameterised agent S below. The parameter t is the tree to be transmitted, c is the channel the tree is sent on, and w is a channel used for initiating the transition of the right sub-tree after the transmission of the left sub-tree has finished. The projection functions fst, snd and thrd are used to access the elements of a tuple.

$$
\begin{array}{l}
S(t,c,w) \stackrel{\text{def}}{=} \\
\quad \texttt{if } t = \mathsf{Leaf} \texttt{ then} \\
\qquad c \triangleleft \mathsf{Leaf}.\overline{w}\langle c \rangle.\mathbf{0} \\
\quad \texttt{else} \\
\qquad c \triangleleft \mathsf{Node}. \\
\qquad \overline{c}\langle \mathsf{fst(t)} \rangle. \\
\qquad (\nu n)(S(\mathsf{snd}(t), c, n) \mid n(x).S(\mathsf{thrd}(t), c, w))
\end{array}
$$

If we analyse the S process, the endpoint type of c is T_C, which is the same type as was described in Section 3.1. The type describes how a single node is transmitted, if it is a leaf node we do not do anything, if it is internal node then the value is transmitted and then the sub-trees of the node.

$$
\begin{array}{l}
T_C = \mu z. \oplus \{ \\
\quad \mathsf{Leaf} : \mathsf{lin\ skip} \\
\quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; z; z \\
\}
\end{array}
$$

The process below receives the transmitted tree preserving the original structure. Through similar analysis as on the sending end, we can confirm that the endpoint type of c in this process is $\overline{T_C}$.

$$
\begin{array}{l}
R(c,w) \stackrel{\text{def}}{=} c \triangleright \{ \\
\quad \mathsf{Leaf} : \overline{w}\langle c \rangle.\mathbf{0} \\
\quad \mathsf{Node} : (\nu n)(c(x).R(c,n) \mid n(x).R(c,w)) \\
\}
\end{array}
$$

We can now create a process P that transmits a tree on the c channel.

$$P = S(\mathsf{Tree}(1, \mathsf{Leaf}, \mathsf{Leaf}), c, w) \mid R(c,w)$$

With an addition to P, we can create a process P' that reuses c for transmitting an integer after transmitting the tree.

$$P' = [S(\mathsf{Tree}(1, \mathsf{Leaf}, \mathsf{Leaf}), c, w) \mid R(c,v)] \mid \left[v(c').c'(x).\mathbf{0} \mid w(c'').\overline{c''}\langle 1 \rangle.\mathbf{0}\right]$$

The type of c after expanding the recursive type is now $(T'_C, \overline{T'_C})$ where

$$
\begin{array}{l}
T'_C = \mathsf{lin}\ \oplus \{ \\
\quad \mathsf{Leaf} : \mathsf{lin\ skip} \\
\quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; \\
\qquad \mu z. \oplus \{\mathsf{Leaf} : \mathsf{lin\ skip} \quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; z; z\}; \\
\qquad \mu z. \oplus \{\mathsf{Leaf} : \mathsf{lin\ skip} \quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; z; z\} \\
\}; \mathsf{lin}\ !\mathsf{Int}
\end{array}
$$

The last step is what motivates type equivalence; we would like to have a *distributive law* that allows the output to be moved into the select type, as illustrated below.

$$
\begin{array}{l}
T''_C = \mathsf{lin}\ \oplus \{ \\
\quad \mathsf{Leaf} : \mathsf{lin\ skip}; \mathsf{lin}\ !\mathsf{Int} \\
\quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; \\
\qquad \mu z. \oplus \{\mathsf{Leaf} : \mathsf{lin\ skip} \quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; z; z\}; \\
\qquad \mu z. \oplus \{\mathsf{Leaf} : \mathsf{lin\ skip} \quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; z; z\}; \\
\qquad \mathsf{lin}\ !\mathsf{Int} \\
\}
\end{array}
$$

As the typing rules in our applied pi-calculus are defined on the type transitions instead of the structure of a type, the distributive law is not essential in our calculus, since we require that the type exhibit specific behaviour instead of having a specific structure, but the example shows how type equivalence can be used to tell if two types can be used interchangeably. In the next section we will introduce a method for checking if two types are equivalent, and we will return to Example 1, to check that it is indeed the case that T''_C is equivalent to T'_C.

6.2 Type bisimilarity

As described in Section 3.3, our types are highly reminiscent of BPA. For BPA expressions, bisimulation is used to prove that two processes exhibit the same behaviour. We now extend bisimulation to work on types as well. The definition follows from the definition of bisimulation in [2, p. 37].

Definition 1. (Type bisimulation) A binary relation $\mathcal{R}$ between endpoint types is a type bisimulation iff whenever $T_{E_1} \mathcal{R}\ T_{E_2}$:

- $Q(T_{E_1}) = Q(T_{E_2})$
- $\forall \lambda$ if $T_{E_1} \xrightarrow{\lambda} T'_{E_1}$ then $\exists T'_{E_2}$ such that $T_{E_2} \xrightarrow{\lambda} T'_{E_2}$ and $T'_{E_1} \mathcal{R}\ T'_{E_2}$
- $\forall \lambda$ if $T_{E_2} \xrightarrow{\lambda} T'_{E_2}$ then $\exists T'_{E_1}$ such that $T_{E_1} \xrightarrow{\lambda} T'_{E_1}$ and $T'_{E_1} \mathcal{R}\ T'_{E_2}$

We write that $T_{E_1} \sim T_{E_2}$ if $T_{E_1} \mathcal{R} T_{E_2}$ for some type bisimulation and then say that T_{E_1} and T_{E_2} are *type bisimilar*.

Type bisimilar endpoint types will exhibit the same behaviour. In other words, for a type T_E, any type T'_E where $T_E \sim T'_E$, T'_E can be used instead of T_E, without introducing communication errors.

Example 2. (A distributive law) Consider the types T'_C and T''_C from Example 1.

We can use type bisimulation to show that these two types describe the same communication behaviour on a channel. To do so, we must provide a bisimulation that shows that T'_C and T''_C are type bisimilar.

Let R be a relation over endpoint types. Let R be the symmetric closure of

$$\{(T'_C, T''_C), ((\mathsf{lin}\ !\mathsf{int}; r; r); \mathsf{lin}\ !\mathsf{int}, \mathsf{lin}\ !\mathsf{int}; r; r; \mathsf{lin}\ !\mathsf{int})\} \cup \{(T_E, T_E) \mid \forall T_E \in \mathcal{T}\}$$

where r is the term $\mu z. \oplus\{\mathsf{Leaf} : \mathsf{lin}\ \mathsf{skip} \quad \mathsf{Node} : \mathsf{lin}\ !\mathsf{Int}; z; z\}$.

In fact, we can generalise this result and prove the distributive law for both select and branch.

Lemma 2. *Let $\star$ be $\oplus$ or $\&$. Then $q \star \{l_i : T_{E_i}\}_{i \in I}; T_E \sim q \star \{l_i : T_{E_i}; T_E\}_{i \in I}$*

Proof. Let R be a relation between types. We must show that R is a bisimulation of $q \star \{l_i : T_{E_i}\}_{i \in I}; T_E$ and $q \star \{l_i : T_{E_i}; T_E\}_{i \in I}$. Define R as the symmetric closure of

$$\{(q \star \{l_i : T_{E_i}\}_{i \in I}; T_E, q \star \{l_i : T_{E_i}; T_E\}_{i \in I})\} \cup \{(T_E, T_E) \mid \forall T_E \in \mathcal{T}\}$$

From the first requirement for a type bisimulation we have that $Q(q \star \{l_i : T_{E_i}\}_{i \in I}; T_E) = Q(q \star \{l_i : T_{E_i}; T_E\}_{i \in I}))$, this is trivially fulfilled as $q = q$. By the (SEQ) rule we have that the transitions of a sequential compositions are those of the left-hand side of the operator. So the transitions of $q \star \{l_i : T_{E_i}\}_{i \in I}; T_E$ are $q \star \{l_i : T_{E_i}\}_{i \in I}; T_E \xrightarrow{\diamond l_i} T_{E_i}; T_E$, where $\diamond \in \{\triangleleft, \triangleright\}$. The available transitions for $q \star \{l_i : T_{E_i}; T_E\}_{i \in I}$ are $q \star \{l_i : T_{E_i}; T_E\}_{i \in I} \xrightarrow{\diamond l_i} T_{E_i}; T_E$. Since R is reflexive, we have that $(T_{E_i}; T_E, T_{E_i}; T_E) \in R$. So any transition taken by one of the types can be matched by the other to end up with syntactically equivalent types. Since the types are syntactically equivalent, all further transitions can be matched by any of the two types, and all further type pairs will be in R. This means that R is a type bisimulation, and that $q \star \{l_i : T_{E_i}\}_{i \in I}; T_E \sim q \star \{l_i : T_{E_i}; T_E\}_{i \in I}$. □

6.3 A quotiented session type system

We now define a quotiented session type system whose types are equivalence classes of session types from the the already existing type system. We define transitions between equivalence classes instead of endpoint types as follows.

Definition 2. (Equivalence Classes) Let $|T_E|$ be the equivalence class of T_E given by:

$$|T_E| = \{T'_E \in \mathcal{T} \mid T'_E \sim T_E\}$$

We denote a transition between equivalence classes with action λ as $|T_{E_1}| \xrightarrow{\lambda} |T_{E_2}|$.

Lemma 3. $|T_E| \xrightarrow{\lambda} |T'_E|$ *iff* $\forall T_{E_1} \in |T_E|\ \exists T'_{E_1} \in |T'_E|$ *such that* $T_{E_1} \xrightarrow{\lambda} T'_{E_1}$

Proof. By the properties of type bisimilarity that states that two bisimilar types will evolve to bisimilar types for all possible transitions. □

We use Lemma 3 to define a new type system that is defined on equivalence classes of types from the previous type system. In Section 3.3 the transitions of endpoint types were of the form $T_E \xrightarrow{\lambda} T'_E$. In the new type system the transitions are of the form $|T_E| \xrightarrow{\lambda} |T'_E|$. The transition rules from Table 3 still apply to the new type system, but where all endpoint types T_E have been replaced with $|T_E|$. For example, the (SELECT) rule from Table 3 would in the new type system be (13). We also expand the Q function on equivalence classes to be the result of applying Q to any witness of the equivalence class.

$$\text{(SELECT)} \qquad \frac{q \sqsubseteq Q(|T_{E_k}|)}{|q \oplus \{l_i : |T_{E_i}|\}_{i \in I}| \xrightarrow{\lhd l_k} |T_{E_k}|} \tag{13}$$

The typing rules in the new type system would also be the rules from Tables 4, 7 and 8, where endpoint types T_E have been replaced with equivalence classes $|T_E|$. In 14 the (INPUT) rule from Table 8 has been changed to fit the new type system.

$$\text{(INPUT)} \qquad \frac{\Gamma_1 \vdash n : |T_E| \qquad |T_E| \xrightarrow{?T} |T'_E| \qquad \Gamma_2, x : T + n : |T'_E| \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash n(x).P} \tag{14}$$

The generic type system from [7] depends on the transitions available to types. We have already shown that the previous type system is an instance of generic type systems. From Lemma 3, we can see that equivalence classes have the exact same transitions as the old endpoint types had. From these two results we can see that the type system defined on equivalence classes is an instance of the generic type system as well, allowing us to retain previously obtained results of fidelity and well typed internal actions from Section 4. In conclusion, this gives us a type system where type equivalence is a trivial property, since two endpoint types of the old type system would be the same type in the new type system.

7 Conclusion

In this paper we have considered a "low-level" applied pi-calculus that allows composite terms to be built but only allows for passing names and nullary function symbols. In this setting, we introduced a type system for the applied pi-calculus based on the work on context-free session types by Thiemann and Vasconcelos in [9] and on the work on qualified session types by Vasconcelos in [10]. The type system

is a context-free session type system with qualifiers. The type system is an instance of the psi-calculus type system introduced in [7], and this allows us to establish a fidelity result about the type system that ensures that a well typed process continues to be well typed, and without communication errors, until termination.

The type system has a notion of type equivalence defined by introducing type bisimilarity. Using it, we then get a type system of equivalence classes, for which type equivalence is a natural part of the type system itself.

The current focus is to deal with the decidability of type equivalence. In [9] Thiemann and Vasconcelos show the decidability of type equivalence using a transformation from their types to guarded BPA expressions, for which bisimilarity is decidable. As already discussed in this article, our types are very close to BPA, in which case we should be able to achieve the same results about decidability more directly. In [5] an algorithm is presented for deciding bisimilarity for normed context free processes in polynomial time, and we conjecture that this algorithm can be adapted to our setting for checking type bisimilarity. Here, it would be important to find a characterization of the class of applied pi processes that can be typed using normed session types only.

References

[1] Martín Abadi, Bruno Blanchet & Cédric Fournet (2018): *The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication*. *J. ACM* 65(1), pp. 1:1–1:41, doi:10.1145/3127586.

[2] Luca Aceto, Anna Ingólfsdóttir, Jiri Srba & Kim Guldstrand Larsen (2007): *Reactive systems : modelling, specification and verification*. Cambridge University Press, Cambridge, UK New York, doi:10.1017/CBO9780511814105.

[3] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2011): *Psi-calculi: a framework for mobile processes with nominal data and logic*. *Logical Methods in Computer Science* 7(1), doi:10.2168/LMCS-7(1:11)2011.

[4] J. A. Bergstra & J. W. Klop (1989): *Process theory based on bisimulation semantics*. In J. W. de Bakker, W. P. de Roever & G. Rozenberg, editors: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 50–122, doi:10.1007/BFb0013021.

[5] Yoram Hirshfeld & Faron Moller (1994): *A Fast Algorithm for Deciding Bisimilarity of Normed Context-Free Processes*. In: *CONCUR'94: Concurrency Theory*, Springer Berlin Heidelberg, pp. 48–63, doi:10.1007/978-3-540-48654-1_5.

[6] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, Springer-Verlag, London, UK, UK, pp. 122–138, doi:10.1007/bfb0053567.

[7] Hans Hüttel (2016): *Binary Session Types for Psi-Calculi*. In Atsushi Igarashi, editor: *Programming Languages and Systems*, Springer International Publishing, Cham, pp. 96–115, doi:10.1007/978-3-319-47958-3_6.

[8] Robin Milner (1999): *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[9] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free Session Types*. *SIGPLAN Not.* 51(9), pp. 462–475, doi:10.1145/3022670.2951926.

[10] Vasco T. Vasconcelos (2011): *Sessions, from Types to Programming Languages*. *Bulletin of the European Association for Theoretical Computer Science* 103, pp. 54–73, doi:10.1.1.228.6236. Available at http://eatcs.org/images/bulletin/beatcs103.pdf.