

# Dependent Session Types

Hanwen Wu<sup>1</sup> and Hongwei Xi<sup>2</sup>

<sup>1</sup> Boston University  
hwwu@cs.bu.edu

<sup>2</sup> Boston University  
hwxi@cs.bu.edu

---

## Abstract

Session types offer a type-based discipline for enforcing that communication protocols be correctly followed in distributed programmes. Our previous work has formalized simple session types in the settings of multi-threaded  $\lambda$ -calculus with linear types. However, there are still protocols that can't be described precisely in simple session types. In this work, we extend our previous results to present *dependent session types* that supports quantification of session types over static terms. We provide linearity and duality guarantees natively and statically by our type system without runtime checks or any special encodings. Our technical results include preservation and progress properties. If a program conforms to some dependent session type, the program will adhere to session protocols and will not deadlock. Our formulation is the first dependent session type system to base on  $\lambda$ -calculus. Such formulation is practical to implement, and we describe one of our implementations in Erlang/Elixir.

**1998 ACM Subject Classification** F.1.2 Modes of Computation: Parallelism and concurrency; F.4.1 Mathematical Logic: Lambda calculus and related systems

**Keywords and phrases** session types, linear types, dependent types, lambda calculus

**Digital Object Identifier** [10.4230/LIPIcs.CONCUR.2017.23](https://doi.org/10.4230/LIPIcs.CONCUR.2017.23)

## 1 Introduction

A session is a series of interactions among concurrently running programs. Session types [5, 6, 24, 7] are type disciplines safe-guarding such interactions by assigning session types to communication channels so that all parties can use the channels in perfect harmony. Recent works [27, 26, 2, 4, 3] have established a form of Curry-Howard correspondence where propositions in some logic are session types for terms in some variants of  $\pi$ -calculus [13, 14]. [25] pushed it further to account for dependent session types by incorporating quantifiers.

Instead of  $\pi$ -calculus, it is also possible to formulate session types in the settings of  $\lambda$ -calculus [11, 34]. Such a formulation is closer to concrete implementations. This paper extends [34] and formulates dependent session types.

More specifically, the formulation is based on Applied Type Systems ( $\mathcal{ATS}$ [30, 28]), a type system supporting dependent types (of DML-style [33]), linear types, and programming with theorem proving.  $\mathcal{ATS}$  takes a layered approach to dependent types in which *statics*, where types are formed and reasoned about, are completely separated from *dynamics*, where programs are constructed and evaluated. Based on  $\mathcal{ATS}$ , session protocols are then captured by extending statics with session types (static terms of sort *stype*), while communication channels are *linear* dynamic values whose types are *indexed* by such session protocols.

A very important difference of our formulation as compared to other similar works, say [25], is that our session types describe the intended behavior *globally*, instead of using a polarized presentation where dual session types are used to describe dual endpoints of a



© Hanwen Wu and Hongwei Xi;  
licensed under Creative Commons License CC-BY

28th International Conference on Concurrency Theory.

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:20



Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

channel *locally*. This global treatment of session types also applies to quantifiers. We shall see the handling of quantifiers from the following example.

► **Example 1 (Array)**. One can safely send an array by sending a length  $n$  first, then followed by  $n$  messages for  $n$  elements of the array. Such a channel can be encoded in the following session types.

$$\begin{aligned} \text{repeat}(\tau:\text{type}, n:\text{int}) &::= \text{ite}(n > 0, \text{msg}(0, \tau) :: \text{repeat}(\tau, n - 1), \text{end}(0)) \\ \text{array}(\tau:\text{type}) &::= \text{quan}(0, \lambda n:\text{int}.\text{msg}(0, \text{int}(n)) :: \text{repeat}(\tau, n)) \end{aligned}$$

where **repeat** is a recursive session type written in direct style, and its desugared version is as follows,

$$\text{repeat}(\tau:\text{vtype}) ::= \text{fix}(\lambda p:(\text{int} \rightarrow \text{stype}).\lambda n:\text{int}.\text{ite}(n > 0, \text{msg}(0, \tau) :: p(n - 1), \text{end}(0)))$$

Note that **repeat** and **array** are type schemes, parameterized by a type  $\tau$ . Thus,  $\tau$  is not bound by the **fix**, an encoding of the fix-point operator. Let's use *roles*, 0 (server) and 1 (client), to represent the two participants in a session. Let's fix some type  $\tau$ . **repeat**( $\tau, n$ ) then says, if  $n > 0$  is true, the session proceeds to allow sending of a value of type  $\tau$  from party 0 ( $\text{msg}(0, \tau)$ ), then proceeds as **repeat**( $n - 1$ ). If  $n > 0$  is false, the session can only be terminated by party 0 (**end**(0)). Similarly, **array** says, party 0 is to send an integer  $n$  followed by  $n$  repeated messages described by **repeat**( $\tau, n$ ), and that this session type is universally quantified at party 0, and dually, existentially quantified at party 1, denoted by a session type constructor **quan**(0, ·). Therefore, the server side can be programmed as follows, roughly using the syntax of ATS, a ML-like language based on *ATS*,

```
fun server {a:type} {n:nat}
  (ch:chan(0,array(a)), data:arrref(a,n), len:int(n)): void = let
  prval () = unify ch (* locally interprets the quantifier *)
  val () = send (ch, len)
  fun loop {a:type} {n,m:nat|n<=m}
    (ch:chan(0,repeat(a,n)), x:int(n), data:arrref(a,m), len:int(m)): void =
    if x = 0 then let prval () = recurse ch
                  prval () = itef ch
                  in close ch end
    else let prval () = recurse ch
            prval () = itet ch
            val () = send (ch, data[len-x])
            in loop (ch, x-1, data, len) end
  in loop (ch, len, data, len) end
```

And its type is

$$\text{server} : \forall \tau:\text{type}.\forall n:\text{nat}.\text{chan}(0, \text{array}(\tau)), \text{arrref}(\tau, n), \text{int}(n) \rightarrow 1$$

Here, **ch** is the endpoint of the communication channel at party 0, denoted as **chan**(0, **array**( $a$ )). **data** is the array to be sent, whose type is indexed by the type of elements and the length of array. **len** is then length of array, whose type is a singleton integer that equals the length of **data**. The details of syntax will become clear in later sections. We will focus only on some key aspects now.

The linear type constructor, **chan**, will construct a linear type **chan**( $r, \pi$ ) given a role  $r$  and a global session type  $\pi$ . The combination of  $r$  and  $\pi$  is where a global session

type gets “projected” locally. This can be used to type an *endpoint* of a channel at party  $r$ . As **array** is globally quantified by session type constructor **quan**, we need to locally interpret it at party 0, by calling a session API **unify** (see Figure 7) which essentially turns  $\mathbf{chan}(0, \mathbf{quan}(0, \lambda n: \mathbf{int}.\mathbf{msg}(0, \mathbf{int}(n)) :: \mathbf{repeat}(\tau, n))$  into

$$\forall n: \mathbf{int}.\mathbf{chan}(0, \mathbf{msg}(0, \mathbf{int}(n)) :: \mathbf{repeat}(\tau, n))$$

for use with other session API, e.g. **send**. The *guard* in the signature of **unify**,  $r = r_0$ , specifies that, for any  $\mathbf{quan}(r_0, \cdot)$  at endpoint  $\mathbf{chan}(r, \cdot)$ , only when  $r = r_0$  is true that **unify** can be invoked to turn  $\mathbf{chan}(r, \mathbf{quan}(r_0, \cdot))$  into  $\forall a: \sigma.\mathbf{chan}(r, \cdot)$ . Dually, the client program will have the following type,

$$\mathbf{client} : \forall \tau: \mathbf{type}.\mathbf{chan}(1, \mathbf{array}(\tau)) \rightarrow \exists n: \mathbf{nat}.\mathbf{int}(n) \times \mathbf{arrref}(\tau, n)$$

and before the client can use the channel to receive the length of the array, it has to locally interpret **quan** at party 1, by calling **exify** (see Figure 7) whose guard is  $r \neq r_0$ , which is the dual of **unify** since roles can only be 0 or 1 in a binary session. It will turn the endpoint into

$$\exists n: \mathbf{int}.\mathbf{chan}(1, \mathbf{msg}(0, \mathbf{int}(n)) :: \mathbf{repeat}(\tau, n))$$

After such local interpretations, it is left to our intermediate language  $\mathcal{L}_{\forall, \exists}$ , multi-threaded  $\lambda$ -calculus with linear types and dependent types, to interpret the quantifiers. Essentially, an universally quantified endpoint *inputs* an instance from the user to eliminate the quantifier and carry out substitution, while an existentially quantified endpoint outputs the witness to the user to eliminate the quantifier. Note that the user of an endpoint is the process holding such endpoint, so “inputs from the user” means the user writes a program to *send* a value using the endpoint. Such twist is found in other works as well, e.g. [27, 26].

The main contribution of this paper is the formulation of dependent session types in the settings of  $\lambda$ -calculus, which is a first to the best of our knowledge. Our system of session types supports quantification over static terms, recursions, and uses unpolarized presentation. Our technical results include preservation and progress properties, which indicates session fidelity and deadlock-freeness. We also present the implementation of dependent session types, which is also a first.

The following sections are organised as follows. Section 2 briefly setup multi-threaded  $\lambda$ -calculus with linear types, denoted as  $\mathcal{L}_0$ . Section 3 introduces *predicativization* to extend  $\mathcal{L}_0$  into multi-threaded  $\lambda$ -calculus with dependent types and linear types, denoted as  $\mathcal{L}_{\forall, \exists}$ . Section 4 further extends  $\mathcal{L}_{\forall, \exists}$  to formulate dependent session types as  $\mathcal{L}_{\forall, \exists}^\pi$ . Section 5 describes technical details of our implementations. Section 6 demonstrates the benefits of dependent session types through examples. We then mention extensions (multi-party sessions, polymorphism, etc) in Section 7, related works in Section 8 and finally conclude in Section 9.

## 2 Multi-threaded $\lambda$ -calculus with Linear Types

The formulation of multi-threaded  $\lambda$ -calculus with linear types is largely standard and follows exactly from our previous work [34] except for some minor cosmetic changes. Therefore, we only present it very briefly and refer the readers to our prior work for details.

### 2.1 Syntax

The syntax is shown in Figure 1 which is mostly standard.  $\delta/\hat{\delta}$  are non-linear/linear base types. Note that a type  $\tau$  is also a linear type  $\hat{\tau}$ , but it is not regarded as a *true* linear type.

■ **Figure 1** Syntax of Multi-threaded  $\lambda$ -calculus with Linear Types

types	$\tau ::= \delta \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2$
vtypes	$\hat{\tau} ::= \hat{\delta} \mid \tau \mid \hat{\tau}_1 \otimes \hat{\tau}_2 \mid \hat{\tau}_1 \multimap \hat{\tau}_2$
dynamic constants	$dcc ::= dcc \mid dcf$
dynamic terms	$e ::= x \mid dcc(\vec{e}) \mid dcr \mid \langle \rangle \mid \langle e_1, e_2 \rangle \mid$ $\text{let } \langle x_1, x_2 \rangle = e_1 \text{ in } e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid$ $\text{fst}(e) \mid \text{snd}(e) \mid \text{lam } x.e \mid \text{app}(e_1, e_2)$
dynamic values	$v ::= x \mid dcc(\vec{v}) \mid \langle \rangle \mid \langle v_1, v_2 \rangle \mid \text{lam } x.e$
dynamic type context	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
dynamic vtype context	$\Delta ::= \emptyset \mid \Delta, x : \hat{\tau}$
dynamic signature	$\mathcal{S} ::= \emptyset \mid \mathcal{S}, dcc : (\vec{\tau}) \Rightarrow \tau \mid \mathcal{S}, dcr : (\vec{\tau}) \Rightarrow \hat{\tau}$
dynamic substitutions	$\theta ::= [] \mid \theta[x \mapsto v]$
pools	$\Pi ::= [] \mid \Pi[t \mapsto e]$

$dcc/dcf$  are dynamic constant constructors/functions.  $dcr$  are dynamic constant resources that are treated *linearly*.  $\mathcal{S}$  are dynamic signatures that assign types to dynamic constants, and these types are called *c-types*. Note that  $\vec{\tau}$  stands for a possibly empty sequence of  $\cdot$ , i.e.  $\vec{e}$  is a possibly empty sequence of dynamic terms.  $dcc(\vec{e})$  is a term of type  $\tau$  if  $dcc$  is a constant of c-type  $(\tau_1, \dots, \tau_n) \Rightarrow \tau$  in  $\mathcal{S}$  and for each  $e_i (1 \leq i \leq n)$  in  $\vec{e}$ ,  $e_i$  has type  $\tau_i$ .

We use  $[]$  for the empty mapping and  $[a_1, \dots, a_n \mapsto b_1, \dots, b_n]$  for a mapping that maps  $a_i$  to  $b_i$  for  $1 \leq i \leq n$ , in which case we write  $m(a_i)$  to mean  $b_i$ . We use  $\text{dom}(m)$  for the domain of a mapping  $m$ . If  $a \notin \text{dom}(m)$ , then  $m[a \mapsto b]$  means to extend  $m$  with a new link from  $a$  to  $b$ . We also use  $m \setminus a$  to mean the mapping obtained by removing  $a$  from  $\text{dom}(m)$ , and  $m[a := b]$  to mean  $(m \setminus a)[a \mapsto b]$ . Substitution  $\theta$  is a mapping from variables to dynamic values. We write  $e[\theta]$  for the result of applying  $\theta$  to  $e$ . Pool  $\Pi$  is a mapping from thread identifiers  $t$  (represented as natural numbers) to closed dynamic expressions such that  $0 \in \text{dom}(\Pi)$ . We use  $\Pi(t), t \in \text{dom}(\Pi)$  to refer to a thread in  $\Pi$  whose thread identifier is  $t$ . We use  $\Pi(0)$  for the main thread.

Typing contexts are divided into a non-linear part  $\Gamma$  and a linear part  $\Delta$ . They are intuitionistic meaning that it is required that each variable occurs at most once in a non-linear context  $\Gamma$  or a linear context  $\Delta$ . Given  $\Gamma_1, \Gamma_2$  s.t.  $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$ , we write  $(\Gamma_1, \Gamma_2)$  for the union of the two. The same notion also applies to linear context  $\Delta$ . Given non-linear context  $\Gamma$  and linear context  $\Delta$ , we can form a combined context  $(\Gamma; \Delta)$  when  $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$ . Given  $(\Gamma; \Delta)$ , we may write  $(\Gamma; \Delta), x : \hat{\tau}$  for either  $(\Gamma; \Delta, x : \hat{\tau})$  or  $(\Gamma, x : \hat{\tau}; \Delta)$  if  $\hat{\tau}$  is indeed a non-linear type.

Besides integers and booleans, we also assume a constant function `thread_create` in  $dcc$  whose c-type in  $\mathcal{S}$  is  $(\mathbf{1} \multimap \mathbf{1}) \Rightarrow \mathbf{1}$ . A function of type  $\mathbf{1} \multimap \mathbf{1}$  takes no argument and returns no result (if it terminates). Since it is a true linear function, it contains no resources and can be invoked exactly once. Intuitively, `thread_create` creates a thread that evaluates the linear function. Its semantic is to be formally introduced later.

To manage resources, we follow [34] and define  $\rho(\cdot)$  (Figure 9) to compute the multiset (bag) of constant resources in a given expression and  $\mathcal{R}$  (RES in [34]) to range over such multisets of resources. We say  $R$  is valid if  $R \in \mathcal{R}$  holds. Intuitively,  $\mathcal{R}$  can be thought of like all the resources of all the programs and  $R$  the resources of a single program. We need to make sure that resource allocation to different programs is consistent in  $\mathcal{R}$ . For precise

definitions, please refer to our prior work.

## 2.2 Semantics

Typing rules are the same as [34], and we push it to Figure 10 in the appendix. The c-type judgment based on the signature is of the form  $\mathcal{S} \models e : \hat{\tau}$ . A typing judgment is of the form  $\Gamma; \Delta \vdash e : \hat{\tau}$  which is standard. By inspecting the rules in Figure 10, we can readily see that a closed value cannot contain resources if it can be assigned a non-linear type  $\tau$ . The *Lemma of Canonical Forms* and the *Lemma of Substitution* are the same as our previous work ([34] Lemma 2.2 and Lemma 2.3), we thus omit them completely.

$\mathcal{L}_0$  has a call-by-value semantic, and the definition of evaluation context ( $E$ ), redex, and reducts are completely standard and are the same as our previous work. We thus omit the details and present only reduction on pools and properties of  $\mathcal{L}_0$ . Given pools  $\Pi_1, \Pi_2$ , we define *parallel reductions on pools*  $\Pi_1 \rightarrow \Pi_2$  as follows,

$$\frac{e_1 \rightarrow e_2}{\Pi[t \mapsto e_1] \rightarrow \Pi[t \mapsto e_2]} \text{pr0} \quad \frac{t > 0}{\Pi[t \mapsto \langle \rangle] \rightarrow \Pi} \text{pr2}$$

$$\frac{\Pi(t) = E[\text{thread\_create}(\text{lam } x.e)]}{\Pi \rightarrow \Pi[t := E[\langle \rangle][t' \mapsto \text{app}(\text{lam } x.e, \langle \rangle)]} \text{pr1}$$

► **Theorem 2** (Subject Reduction on Pools). *Assume  $\emptyset; \emptyset \vdash \Pi_1 : \hat{\tau}$  is derivable and  $\Pi_1 \rightarrow \Pi_2$  holds for some  $\Pi_2$  satisfying  $\rho(\Pi_2) \in \mathcal{R}$ . Then  $\emptyset; \emptyset \vdash \Pi_2 : \hat{\tau}$  is also derivable.*

► **Theorem 3** (Progress Property on Pools). *Assume that  $\emptyset; \emptyset \vdash \Pi_1 : \hat{\tau}$  is derivable. Then we have*

- $\Pi_1$  is a singleton mapping  $[0 \mapsto v]$  for some value  $v$ , or
- $\Pi_1 \rightarrow \Pi_2$  holds for some  $\Pi_2$  s.t.  $\rho(\Pi_2) \in \mathcal{R}$ .

► **Theorem 4** (Soundness of  $\mathcal{L}_0$ ). *Assume that  $\emptyset; \emptyset \vdash \Pi_1 : \hat{\tau}$  is derivable. Then for any  $\Pi_2$ ,  $\Pi_1 \rightarrow^* \Pi_2$  implies that either  $\Pi_2$  is a singleton mapping  $[0 \mapsto v]$  for some value  $v$  or  $\Pi_2 \rightarrow \Pi_3$  for some  $\Pi_3$  satisfying  $\rho(\Pi_3) \in \mathcal{R}$ , where  $\rightarrow^*$  is the transitive and reflective closure of  $\rightarrow$ .*

**Proof.** Follows directly from Theorem 2 and Theorem 3. ◀

## 3 Predicativization

In this section, we extremely briefly describe an approach to extend  $\mathcal{L}_0$  to support both universally and existentially quantified types. Such process is *predicativization* and is mostly standard in the framework of *ATS* [30]. Predicativization is extensively described in [29, 33, 31], and has been employed in several other papers based on *ATS*, e.g. [23, 22]. We thus only summarize the process to prepare for the development of  $\mathcal{L}_{\forall, \exists}^\pi$ , and omit any technical details.

As an applied type system,  $\mathcal{L}_{\forall, \exists}$  is layered into *statics* and *dynamics*. The dynamics of  $\mathcal{L}_{\forall, \exists}$  is based on  $\mathcal{L}_0$ , while the statics will be a newly introduced layer underlying  $\mathcal{L}_0$ . The predicativization process concerns mostly about formalizing the type index language while maintaining the dynamic semantics of  $\mathcal{L}_0$ , and reducing type equality problems into constraint solving problems w.r.t. some constraint domain, such as integer arithmetic. General steps of predicativization involve the followings:

- Formalizing statics, the language of type index. This involves its syntax, sorting rules, and specifically, non-linear type/linear type formation rules, etc.
- Formalizing type equality in terms of subtyping relations and regular constraint relations.

■ **Figure 2** Syntax of Statics

base sorts	$b$	$::=$	$int \mid bool \mid type \mid vtype$
sorts	$\sigma$	$::=$	$b \mid \sigma_1 \rightarrow \sigma_2$
static constants	$scx$	$::=$	$scc \mid scf$
static terms	$s$	$::=$	$a \mid scx(\vec{s}) \mid \lambda a:\sigma.s \mid s_1(s_2)$
static context	$\Sigma$	$::=$	$\emptyset \mid \Sigma, a : \sigma$
static signature	$\mathcal{S}$	$::=$	$\emptyset \mid \mathcal{S}, scx : (\vec{\sigma}) \Rightarrow \sigma$
static substitutions	$\theta$	$::=$	$\square \mid \theta[a \mapsto s]$

■ **Figure 3** Types

types	$\tau$	$::=$	$a \mid \delta(\vec{s}) \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid P \supset \tau \mid P \wedge \tau \mid \forall a:\sigma.\tau \mid \exists a:\sigma.\tau$
vtypes	$\hat{\tau}$	$::=$	$\hat{a} \mid \hat{\delta}(\vec{s}) \mid \tau \mid \hat{\tau}_1 \otimes \hat{\tau}_2 \mid \hat{\tau}_1 \multimap \hat{\tau}_2 \mid P \supset \hat{\tau} \mid P \wedge \hat{\tau} \mid \forall a:\sigma.\hat{\tau} \mid \exists a:\sigma.\hat{\tau}$

- Extending dynamics. This involves extending the syntax, typings, evaluation context, and reduction relations to accommodate, for instance, the introduction and elimination of quantifiers.

The language of statics can be regarded as a simply typed  $\lambda$ -calculus. The “types” for static terms are denoted as *sorts* to avoid confusion. The syntax for statics is shown in [Figure 2](#) which is mostly standard. We assume base sorts  $b$  to include *int*, *bool*, *type* for types, and *vtype* for linear types. Non-linear/linear types in the  $\mathcal{L}_{\forall, \exists}$  are now static terms of sorts *type/vtype*, respectively. We reformulate types in the dynamics in [Figure 3](#).

Given a proposition  $P$  (a static term of sort *bool*) and a type  $\tau$ ,  $P \supset \tau$  is a *guarded type*, and  $P \wedge \tau$  is an *asserting type*. Formal definition of guarded types and asserting types can be found in [\[31\]](#). Intuitively, in order to turn a value of type  $P \supset \tau$  into a value of type  $\tau$ , we must establish the proposition  $P$ , thus “guarded”; if a value of type  $P \wedge \tau$  is generated, we can assume that the proposition  $P$  holds, thus “asserting”.

The extended syntax of  $\mathcal{L}_{\forall, \exists}$  over that of  $\mathcal{L}_0$  is given in [Figure 4](#). Typing judgement in  $\mathcal{L}_{\forall, \exists}$  is of the form  $\Sigma; \vec{P}; \Gamma; \Delta \vdash e : \hat{\tau}$  where  $\Sigma$  is sorting environment for static terms and  $\vec{P}$  is a sequence of propositions keeping track of the constraints. We present only some additional typing rules in [Figure 5](#).

We claim that [Theorem 2](#), [Theorem 3](#), and [Theorem 4](#) can be carrier over to  $\mathcal{L}_{\forall, \exists}$  following the proof in [\[31\]](#).

■ **Figure 4** Extended Dynamic Language Syntax

dynamic terms	$e$	$::=$	$\cdots \mid \supset^+(v) \mid \supset^-(e) \mid \wedge(e) \mid \mathbf{let} \ \wedge(x) = e_1 \ \mathbf{in} \ e_2 \mid$ $\forall^+(v) \mid \forall^-(e) \mid \exists(e) \mid \mathbf{let} \ \exists(x) = e_1 \ \mathbf{in} \ e_2$
dynamic values	$v$	$::=$	$\cdots \mid \supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)$

■ **Figure 5** Some Additional Typing Rules of  $\mathcal{L}_{\forall, \exists}$

$$\begin{array}{c}
\frac{\Sigma, a : \sigma; \vec{P}; \Gamma; \Delta \vdash v : \hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \forall^+(v) : \forall a : \sigma. \hat{\tau}} \text{ty-}\forall\text{-intr} \qquad \frac{\Sigma \vdash s : \sigma \quad \Sigma; \vec{P}; \Gamma; \Delta \vdash e : \forall a : \sigma. \hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \forall^-(e) : \hat{\tau}[a \mapsto s]} \text{ty-}\forall\text{-elim} \\
\\
\frac{\Sigma \vdash s : \sigma \quad \Sigma; \vec{P}; \Gamma; \Delta \vdash e : \hat{\tau}[a \mapsto s]}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \exists(e) : \exists a : \sigma. \hat{\tau}} \text{ty-}\exists\text{-intr} \qquad \frac{\Sigma; \vec{P}; \Gamma; \Delta \vdash e_1 : \exists a : \sigma. \hat{\tau}_1 \quad \Sigma, a : \sigma; \vec{P}; (\Gamma; \Delta), x : \hat{\tau}_1 \vdash e_2 : \hat{\tau}_2}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \text{let } \exists(x) = e_1 \text{ in } e_2 : \hat{\tau}_2} \text{ty-}\exists\text{-elim}
\end{array}$$

## 4 Dependent Session Types

Dependent types are types that depend on terms, and they offer much more expressive power for specifying intended behavior of a program through types. A restricted form of dependent types, we call dependent types of DML-style [31], are types that depend on *static* terms. In this section, we will formally develop *dependent session types* (of DML-style), where session types can have quantification over static terms. As a contrived example, imagine we have an equality testing service, that given two integers  $m$  and  $n$ , returns whether they are equal or not as a boolean value. We could describe the protocol as

$$\text{msg}(1, \text{int}) :: \text{msg}(1, \text{int}) :: \text{msg}(0, \text{bool}) :: \text{end}(0)$$

where 0 represents server, 1 represents client,  $\text{msg}(r, a)$  means the party  $r$  will send a (linear) value of type  $a$  to the other side,  $::$  chains together many actions in order, and  $\text{end}(r)$  means the party  $r$  will terminate the session (while the other side waits for the termination). The protocol merely describes the types of input/output but conveys no information about the intended functionality of the service. However, the following dependent session type is much more precise.

$$\begin{aligned}
&\text{quan}(1, \lambda m : \text{int}. \text{quan}(1, \lambda n : \text{int}. \\
&\quad \text{msg}(1, \text{int}(m)) :: \text{msg}(1, \text{int}(n)) :: \text{msg}(0, \text{bool}(m = n)) :: \text{end}(0)))
\end{aligned}$$

where  $\text{quan}$  is a *global* encoding of quantifiers,  $\text{int}$  and  $\text{bool}$  are type constructors (static constants of c-sort  $\text{int} \Rightarrow \text{type}$  and  $\text{bool} \Rightarrow \text{type}$ , respectively), and therefore  $\text{int}(i)$  and  $\text{bool}(b)$  are singleton types representing values that equal  $i$  and  $b$ , respectively. This session type specifies the relation between two integer inputs and the boolean output, that is, given integers  $m$  and  $n$ , the server should send back a boolean value that equals the result of testing  $m = n$ . Such session type forces the server to be only able to implement an equality service on the channel. In the followings, we will formally extend the statics and dynamics of  $\mathcal{L}_{\forall, \exists}$  to formulate  $\mathcal{L}_{\forall, \exists}^\pi$ .

### 4.1 Extending Statics

The syntax of extended statics is given in Figure 6. We add *stype* as a new base sort to represent session types. Session types  $\pi$  are now static terms of sort *stype*. We use  $i$  for static integers and  $b$  for static booleans.  $\text{end}(i)$  means party  $i$  will close the session while the other party will wait for closing. Given linear type  $\hat{\tau}$  and a session type  $\pi$ ,  $\text{msg}(i, \hat{\tau}) :: \pi$  means party  $i$  should send a message to the other party, and then continue as  $\pi$ .  $\text{branch}(i, \pi_1, \pi_2)$  is for branching, where party  $i$  should choose to continue as  $\pi_1$  or  $\pi_2$  while the other party



■ **Figure 6** Syntax of Dependent Session Types

$$\begin{array}{ll}
 \text{base sorts } b & ::= \dots \mid \textit{stype} \\
 \text{types } \pi & ::= \text{end}(i) \mid \text{msg}(i, \hat{\tau}) :: \pi \mid \text{branch}(i, \pi_1, \pi_2) \mid \\
 & \quad \text{ite}(b, \pi_1, \pi_2) \mid \text{quan}(i, \lambda a:\sigma.\pi) \mid \text{fix}(\lambda a:\textit{stype}.\pi)
 \end{array}$$

simply follows the choice. Beyond these basic session type constructs, we have **ite**<sup>1</sup> for conditional branch, **quan** for universal/existential quantification, and **fix** for recursions. Given a static boolean expression, **ite**( $b, \pi_1, \pi_2$ ) represents  $\pi_1$  when  $b$  is  $\top$  (true), or  $\pi_2$  when  $b$  is  $\perp$  (false). Given a static function of sort  $\sigma \rightarrow \textit{stype}$ , **quan**( $i, \lambda a:\sigma.\pi$ ) is interpreted *intuitively*<sup>2</sup> as universally quantified  $\forall a:\sigma.\pi$  by party  $i$ , or as existentially quantified  $\exists a:\sigma.\pi$  by the other party. Note that this is actually a session type *scheme* and we assume the existence of such **quan** for every sort  $\sigma$ . The need for a unified representation of quantifiers, **quan**, is a must since we essentially formulate all session types as *global*, as compared to polarized presentation where session types are all *local*. Given a static function of sort  $\textit{stype} \rightarrow \textit{stype}$ , **fix**( $\lambda a:\textit{stype}.\pi$ ) is an encoding of the fix-point operator that represents the fix-point of the input function. In practice, we may write recursive definitions directly as a syntax sugar as shown in [Example 1](#).

Besides, we also introduce *role* as a subset sort  $\{r:\textit{int} \mid r = 0 \vee r = 1\}$  to represent two parties, server (0) and client (1), involved in a binary session. Note that subset sorts are merely syntax sugars for a guarded/asserting type [\[33\]](#). For instance,  $\forall r:\textit{role}.\text{int}(r)$  is desugared into  $\forall r:\textit{int}.(r = 0 \vee r = 1) \supset \text{int}(r)$ .

We also add the following *linear* type constructor as a static constant,

$$\mathbf{chan} : (\textit{role}, \textit{stype}) \Rightarrow \textit{vtype}$$

that represents a linear channel. Given role  $r$  and session type  $\pi$ , **chan**( $r, \pi$ ) is the endpoint of a channel held by party  $r$ . The channel is governed by the session type  $\pi$ , and the endpoint interprets this session type *locally* at party  $r$ .

## 4.2 Extending Dynamics

We add the following dynamic constant functions, shown in [Figure 7](#), to create, use, and consume linear channels. We will refer to them as *session API* or just the API. **create** is to create a session of two threads, connected via a channel of session type  $\pi$ , and each thread holds an endpoint of the channel. The party  $r_1$  is holding endpoint **chan**( $r_1, \pi$ ) as returned by **create** in the current thread, while the party  $r_2 (\neq r_1)$  is holding endpoint **chan**( $r_2, \pi$ ) in a newly spawned thread evaluating the given *linear* function of type **chan**( $r_2, \pi$ )  $\multimap \mathbf{1}$ . As the (closure) function may contains resources, it must be linear to guarantee that it can be called *exactly once*. The channel endpoint will be consumed in this function as it is linear. **send** is for sending linear values. Given global session type  $\text{msg}(r_0, a) :: \pi$ , its interpretation at party  $r$  where  $r = r_0$  is to send a message of linear type  $a$  then to proceed as  $\pi$ . The **send** function *consumes* the channel, uses the capability of sending denoted by  $\text{msg}(r_0, a)$ , and

<sup>1</sup> Note that **branch** is just a special case of **ite** and we can indeed encode **branch** using **ite**.

<sup>2</sup> This is only intuitively interpreted. Its accurate interpretation should be considered together with an endpoint since  $\pi$  is global. See later sections.



■ **Figure 7** Extended Dynamic Constants

$$\begin{aligned}
\text{create} &: \forall r_1, r_2: \text{role}. \forall \pi: \text{stype}. (r_1 \neq r_2) \supset (\text{chan}(r_2, \pi) \multimap \mathbf{1}) \Rightarrow \text{chan}(r_1, \pi) \\
\text{send} &: \forall r, r_0: \text{role}. \forall \pi: \text{stype}. \forall a: \text{vtype}. (r = r_0) \supset (\text{chan}(r, \text{msg}(r_0, a) :: \pi), a) \Rightarrow \text{chan}(r, \pi) \\
\text{recv} &: \forall r, r_0: \text{role}. \forall \pi: \text{stype}. \forall a: \text{vtype}. (r \neq r_0) \supset (\text{chan}(r, \text{msg}(r_0, a) :: \pi)) \Rightarrow a \otimes \text{chan}(r, \pi) \\
\text{close} &: \forall r, r_0: \text{role}. (r = r_0) \supset (\text{chan}(r, \text{end}(r_0))) \Rightarrow \mathbf{1} \\
\text{wait} &: \forall r, r_0: \text{role}. (r \neq r_0) \supset (\text{chan}(r, \text{end}(r_0))) \Rightarrow \mathbf{1} \\
\text{offer} &: \forall r, r_0: \text{role}. \forall \pi_1, \pi_2: \text{stype}. (r \neq r_0) \supset (\text{chan}(r, \text{branch}(r_0, \pi_1, \pi_2))) \\
&\quad \Rightarrow \exists b: \text{bool}. \text{bool}(b) \otimes \text{chan}(r, \text{ite}(b, \pi_1, \pi_2)) \\
\text{choosel} &: \forall r, r_0: \text{role}. \forall \pi_1, \pi_2: \text{stype}. \\
&\quad (r = r_0) \supset (\text{chan}(r, \text{branch}(r_0, \pi_1, \pi_2))) \Rightarrow \text{chan}(r, \pi_1) \\
\text{chooser} &: \forall r, r_0: \text{role}. \forall \pi_1, \pi_2: \text{stype}. \\
&\quad (r = r_0) \supset (\text{chan}(r, \text{branch}(r_0, \pi_1, \pi_2))) \Rightarrow \text{chan}(r, \pi_2) \\
\text{unify} &: \forall r, r_0: \text{role}. \forall \pi: \text{stype}. \forall f: \sigma \rightarrow \text{stype}. \\
&\quad (r = r_0) \supset \text{chan}(r, \text{quan}(r_0, f)) \Rightarrow \forall s: \sigma. \text{chan}(r, f(s)) \\
\text{exify} &: \forall r, r_0: \text{role}. \forall \pi: \text{stype}. \forall f: \sigma \rightarrow \text{stype}. \\
&\quad (r \neq r_0) \supset \text{chan}(r, \text{quan}(r_0, f)) \Rightarrow \exists s: \sigma. \text{chan}(r, f(s)) \\
\text{itet} &: \forall r: \text{role}. \forall \pi_1, \pi_2: \text{stype}. (\text{chan}(r, \text{ite}(\top, \pi_1, \pi_2))) \Rightarrow \text{chan}(r, \pi_1) \\
\text{itef} &: \forall r: \text{role}. \forall \pi_1, \pi_2: \text{stype}. (\text{chan}(r, \text{ite}(\perp, \pi_1, \pi_2))) \Rightarrow \text{chan}(r, \pi_2) \\
\text{recurse} &: \forall r: \text{role}. \forall f: \text{stype} \rightarrow \text{stype}. (\text{chan}(r, \text{fix}(f))) \Rightarrow \text{chan}(r, f(\text{fix}(f))) \\
\text{cut} &: \forall r_1, r_2: \text{role}. \forall \pi: \text{stype}. (r_1 \neq r_2) \supset (\text{chan}(r_1, \pi), \text{chan}(r_2, \pi)) \Rightarrow \mathbf{1}
\end{aligned}$$

returns another channel of type  $\text{chan}(r, \pi)$ , where the sending capability is now removed. Note that even though we encode it here in the style of continuation, our implementation directly *changes* the type of channel without consuming it. In ATS programming language, it is presented in the following style,

$$\begin{aligned}
\text{send} &: \forall r, r_0: \text{role}. \forall \pi: \text{stype}. \forall a: \text{vtype}. \\
&\quad (r = r_0) \supset (!\text{chan}(r, \text{msg}(r_0, a) :: \pi) \gg \text{chan}(r, \pi), a) \Rightarrow \mathbf{1}
\end{aligned}$$

Dually, the interpretation of  $\text{msg}(r_0, a) :: \pi$  is to receive at party  $r(\neq r_0)$ , implemented by **recv**. Similarly, **close** is for terminating a session while **wait** is waiting for the other side to close. The interpretation of  $\text{branch}(r_0, \pi_1, \pi_2)$  at party  $r(\neq r_0)$  is to offer two choices,  $\pi_1$  and  $\pi_2$ . Therefore, **offer** function will consume the endpoint and return a linear pair of the other party's choice (as a singleton boolean) and the endpoint whose session type is a conditional branch between  $\pi_1, \pi_2$  using the received tag  $b$  as the condition. Dually, **choosel** and **chooser** will choose  $\pi_1$  and  $\pi_2$  respectively by internally sending out a boolean tag to the other party.

**unify** is to interpret  $\text{quan}(r_0, \cdot)$  at party  $r(= r_0)$  as universal quantifier, while **exify** is to interpret it dually as existential quantifier at party  $r(\neq r_0)$ . **itet** and **itef** reduces the conditional branching session type  $\text{ite}(b, \pi_1, \pi_2)$  according to static boolean expression  $b$ . **recurse** unrolls the fix-point encoding. Note that these functions are *proof* functions that merely change the types of endpoints. They have no runtime counterparts and thus can be

eliminated after type checking has passed.

*Duality* is not explicitly encoded as is usually done in session types literature [12, 19, 10]. Instead, we choose to make the duality as general as possible and use a *global* session type  $\pi$  paired with a role  $r$  to guide the local interpretation at endpoint  $r$ . Given that  $r$  can only be 0 or 1, we can define that  $\mathbf{chan}(0, \pi)$  and  $\mathbf{chan}(1, \pi)$  are *dual* endpoints of a channel. Session API come in dual pairs, and the dual usage of dual endpoints are realized by the corresponding session API pairs with the help of guarded types. The typing rules for guarded types will force one endpoint to be only used with one API in the pair while the dual endpoint to be only used with the dual API in the same pair. A crucial indication of such formulation is that we essentially reduce duality checking problem into a simple integer comparison problem, which greatly simplifies our formulation and makes it very straightforward to support multi-party session types (based on [35]). Also, it reduces the number of the dynamic constants in Figure 7 in half by avoiding coercion between so-called input/output types [12]. In our previous work [34], we used a polarized presentation, e.g.  $\mathbf{chanpos}(p)$  and  $\mathbf{channeg}(p)$  where  $p$  is a *local* type. This is similar to  $\mathbf{In}[]/\mathbf{Out}[]$  in [21],  $S_?/S_!$  in [12] Section 6, and *dual/notDual* in [20]. We found this polarized presentation is not suitable for extending to multi-party sessions.

Given *dual* endpoints, **cut** will link together the endpoints by performing *bi-directional forwarding*. In other words, it will send onto one endpoint each received value from the other endpoint. **cut** is often used to implement delegation of service. It can be proven that these two endpoints must belong to *different* channels since otherwise, it will obviously deadlock.

### 4.3 Dynamic Semantics

The dynamic semantics of  $\mathcal{L}_{\forall, \exists}^\pi$  is indeed the same as our prior work except that we have added a branching construct and we use a more general unpolarized presentation. We thus push additional reduction rules on pools in Figure 11 and Figure 12 to the appendix. Note that, as mentioned above, **unify/exify/itet/itef/recurse** do not have any dynamic semantics. The meaning of these rules should be intuitively clear. For instance, **pr-msg** states, if thread  $t_1$  in pool  $\Pi$  is of the form  $E[\mathbf{send}(ch_{i,r_1}, v)]$ , and thread  $t_2$  in pool  $\Pi$  is of the form  $E[\mathbf{recv}(ch_{i,r_2})]$ , then  $\Pi$  can be reduced to another pool where  $t_1$  is replaced by  $E[ch_{i,r_1}]$  and  $t_2$  is replaced by  $E[\langle v, ch_{i,r_2} \rangle]$ .

### 4.4 Soundness of the Type System

While Theorem 2 can be easily established for  $\mathcal{L}_{\forall, \exists}^\pi$ , Theorem 3 is more involved due to the addition of session API. However, based on [30, 33],  $\mathcal{L}_{\forall, \exists}$  and  $\mathcal{L}_{\forall, \exists}^\pi$  are *conservative* extensions of  $\mathcal{L}_0$ , and the deadlock-freeness is proven for  $\mathcal{L}_0$  with channels in [34] using a technique known as *DF-Reducibility*. Thus the same results can be proven for  $\mathcal{L}_{\forall, \exists}^\pi$  using the exact same technique since the dynamic semantics are exactly the same. We thus refer readers to [34, 33] for detailed proofs. We can then establish the same deadlock-freeness guarantee as stated in Lemma 3.1 of [34]

► **Theorem 5** (Subject Reduction of  $\mathcal{L}_{\forall, \exists}^\pi$ ). *Assume that  $\emptyset; \emptyset \vdash \Pi_1 : \hat{\tau}$  is derivable and  $\Pi_1 \rightarrow \Pi_2$  s.t.  $\rho(\Pi_2) \in \mathcal{R}$ . Then  $\emptyset; \emptyset \vdash \Pi_2 : \hat{\tau}$  is derivable.*

► **Theorem 6** (Progress Property of  $\mathcal{L}_{\forall, \exists}^\pi$ ). *Assume that  $\emptyset; \emptyset \vdash \Pi_1 : \hat{\tau}$  is derivable and  $\rho(v)$  contains no channel endpoints for every  $v : \hat{\tau}$ . Then*

- $\Pi_1$  is a singleton mapping  $[0 \mapsto v]$  for some  $v$ , or
- $\Pi_1 \rightarrow \Pi_2$  holds for some  $\Pi_2$  s.t.  $\rho(\Pi_2) \in \mathcal{R}$ .

► **Theorem 7** (Soundness of  $\mathcal{L}_{V,\exists}^\pi$ ). Assume that  $\emptyset; \emptyset \vdash \Pi_1 : \hat{\tau}$  is derivable and  $\rho(v)$  contains no channel endpoints for every  $v : \hat{\tau}$ . Then for any  $\Pi_2$  satisfying  $\rho(\Pi_2) \in \mathcal{R}$ ,  $\Pi_1 \rightarrow^* \Pi_2$  implies either  $\Pi_2$  is a singleton mapping  $[0 \mapsto v]$  for some  $v$ , or  $\Pi_2 \rightarrow \Pi_3$  for some  $\Pi_3$  s.t.  $\rho(\Pi_3) \in \mathcal{R}$ .

## 5 Implementations

Our implementations consist of two parts, a session API library in ATS, and a runtime implementation of the session API (referred to as a *back-end*) in a target language. ATS is a programming language based on *ATS*, and it supports a style of *co-programming* with many target languages by compiling an ATS program into the target language. Its default compilation target is C. For the purpose of this paper, besides a native back-end in ATS/C itself, we also support back-ends in Erlang/Elixir and JavaScript. A session-typed program will be firstly type-checked based on the type system of  $\mathcal{L}_{V,\exists}^\pi$ , and then compiled into a target language (if passed type checking). The compiler/interpreter of the target language will then be invoked to compile/interpret the program together with the corresponding back-end. Although formalized as synchronous sessions (for the sake of simplicity), our implementations can fully support asynchronous communications.

Our session API library in ATS is (almost) a direct translation of those listed in Figure 7, except for some slight syntax differences. For example, `send` is translated into the followings.

```
fun send {r,r0:role|r0==r} {p:stype} {v:vtype}
  (!chan(r,msg(r0,v)::p) >> chan(r,p), v): void
```

where  $\{ \}$  is universal quantification (and  $[ ]$  is existential quantification),  $!$  means *not* to consume a linear value, and  $>>$  means to *change* the linear type after the function returns. As mentioned before, whenever possible, the API will change the types of endpoints directly instead of relying on continuations. There are a couple other minor changes. First, with guarded recursive data types [32] and pattern matching, the API formulates *offer/choose/chooser* in a more uniform way as below.

```
datatype choice (stype, stype, stype) =
  | {p,q:stype} Left (p, p, q) of ()
  | {p,q:stype} Right (q, p, q) of ()
fun offer {r,r0:role|r0!=r} {p,q:stype}
  (!chan(r,branch(r0,p,q)) >> chan(r,s)): [s:stype] choice (s,p,q)
fun choose {r,r0:role|r0==r} {p,q,s:stype}
  (!chan(r,branch(r0,p,q)) >> chan(r,s), choice(s,p,q)): void
```

where *choice* is a linear guarded recursive data type that essentially captures the equality on session types. Also, since it is a linear value, the type-checker will enforce *exhaustive* case analysis on the received choice to consume it.

We briefly mention some technical details below and refer the readers to <http://multirolelogic.org> for pointers to all the source code. Due to space limitation, we assume that the readers are reasonably familiar with these target languages.

### 5.1 Message-passing Back-end in Erlang/Elixir

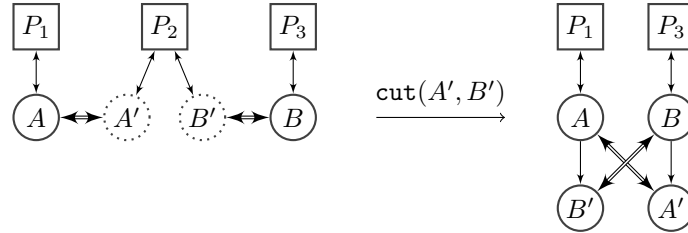
Erlang offers functional distributed programming abilities through its powerful virtual machine. Elixir offers a more friendly syntax and better tooling on top of the same runtime.

In Erlang/Elixir, every process has a unique `pid` (process identifier), and an associated mailbox. Communications are achieved via message-passing asynchronously and can be done across different nodes. In this particular implementation, `choose` and `offer` are implemented as `send` and `receive`, respectively. `close` and `wait` are implemented both to terminate the process directly. This back-end is inherently asynchronous and distributed.

In Erlang/Elixir back-end, a message is represented by a label, a `pid`, a `ref`, and a payload. A channel endpoint is identified through a combination of a `pid` and a `ref`. The message labels are used to identify the kind of messages, e.g. `:send/receive`. The `pid` is used to locate the message's origin, or an endpoint's mailbox. The `ref`'s are globally unique references, generated through a built-in function `make_ref` for every endpoint. The need for `ref` is discussed in [15]. Intuitively speaking, the `ref` acts as a signature of the message and every out-going message is signed using the sending endpoint's own `ref`. Thus it can be used both to distinguish in-session messages from out-of-session messages<sup>3</sup>, and to identify requests from the endpoint's owning process and messages from the dual endpoint.

An endpoint will run a loop in a dedicated process and talk to the owning process through messages-passing. The endpoint loop keeps track of two parameters: `self`, which is its own signature as a `ref`, and `dual`, which is the dual endpoint's `pid` and `ref`. In every iteration, the loop will receive a request from the owning process by pattern matching against messages signed by `self`, and then process the request accordingly. For instance, when the owning process sends a message with label `:receive` signed with `self`, the endpoint will then pattern match against messages in the endpoint's mailbox and block until it finds the first message whose label is `:send` and is signed by the dual endpoint's `ref`, which is `dual.ref`. The found message will then be delivered to the owning process's mailbox, fulfilling the request.

■ **Figure 8** Example `cut` in Erlang/Elixir Back-end



`cut` is implemented as delegation, where `:send` requests are handled as before, but `:receive` requests are delegated to an endpoint involved in a `cut`. Suppose we have dual endpoints  $A:\text{chan}(0,p)/A':\text{chan}(1,p)$  and dual endpoints  $B:\text{chan}(0,p)/B':\text{chan}(1,p)$  of some session type  $p$ , and we are to perform `cut(A', B')`. The owning process  $P_2$  of both  $A'$  and  $B'$ , will send a `:cut` request to  $A'$  and  $B'$ , with a payload of the `pid` and `ref` of  $B'$  and  $A'$ , respectively. The info about  $B'$  will be forwarded to  $A$ , and  $A$  will delegate `:receive` requests to  $B'$ . Similarly, the info about  $A'$  will be forwarded to  $B$ , and  $B$  will delegate `:receive` requests to  $A'$ . A delegated request will change its signature from the original requester's `ref`, to the delegator's `ref`, so that the delegator can still process the request as if the request comes from its owning process. An example is illustrated in Figure 8, where  $\leftrightarrow$  is for endpoint ownership,  $\Leftrightarrow$  connects dual endpoints, and  $\rightarrow$  denotes delegation.

We also have a shard memory implementation in ATS/C which implements our own

<sup>3</sup> This is because that knowing just the `pid` is enough for any process to randomly inject messages to its mailbox.

message queue guarded by locks, and a continuation-based implementation in JavaScript using WebWorker. Please visit the provided website for pointers to the source code.

## 6 Examples

We will show some example dependent session types or programs in the followings. We will assume that the server plays role 0, and the client plays role 1. We will use ATS's ML-like syntax to present the program (after omitting some insignificant details), which can be easily mapped to  $\mathcal{L}_{\forall, \exists}^{\pi}$ . We also use syntax sugar and implementation optimizations described in Section 5 and extensions from Section 7. Again, the source code can be found online through <http://multirolelogic.org>, and all the code can be type-checked, compiled, and executed.

► **Example 8 (Counter).** One can easily define a counter as an integer stream. But more precisely, we can define

$$\text{counter}(n:\text{int}) ::= \text{branch}(1, \text{msg}(0, \text{int}(n)) :: \text{counter}(n+1), \text{end}(1))$$

which says, in every iteration, the client can choose to receive an integer  $n$  and let the session continue from  $n+1$ , or to end the session. On top of **counter**, we can define a service **from** that given an integer  $n$ , returns an endpoint of session type **counter**( $n$ ).

$$\text{from} ::= \text{quan}(1, \lambda n:\text{int}.\text{msg}(1, \text{int}(n)) :: \text{msg}(0, \text{chan}(1, \text{counter}(n))) :: \text{end}(1))$$

Since **chan** is a linear type constructor, a channel can then be sent over another channel just as other linear values, and **send** will consume it. This forms a high order session type. We omit any testing code since it is similar to Example 1. Due to space limitation, we push other examples to Appendix A.

## 7 Extensions

We very briefly describe possible extensions of  $\mathcal{L}_{\forall, \exists}^{\pi}$ . First, it is straitforward to add *general recursion* to our language (not to the session type) as has been done in [34]. We didn't include it just to save space. Second, one can always introduce a *higher-order fix* into session types, such as  $\text{fix}(\lambda f: (\vec{\sigma} \rightarrow \text{stype}). \lambda \vec{a}: \vec{\sigma}. \pi), \vec{s})$  where  $f$  is a static function of sort  $(\vec{\sigma} \rightarrow \text{stype}) \rightarrow \vec{\sigma} \rightarrow \text{stype}$ , and  $\vec{s}$  are static terms of matching sorts  $\vec{\sigma}$ . Correspondingly, we need to introduce another **recurse** to unroll it. More importantly, we can extend  $\mathcal{L}_{\forall, \exists}^{\pi}$  to support *multi-party session types* based on [35]. Roles will be extended from  $\{0, 1\}$  to a larger set of natural numbers, **chan**( $r, \pi$ ) will be extended to **chan**( $R, \pi$ ) where  $R$  is now a *set* of roles, guards in session API will change from  $r = r_0$  to  $r_0 \in R$ , and from  $r \neq r_0$  to  $r_0 \notin R$ . **cut** will be extended to another form based on [35]. We omit any details since it is *very* out-of-scope. Also, both predicative quantification (dependent types) and impredicative quantification (polymorphism) are supported by *ATS*, and our formulation natually supports *polymorphic session types* since **quan** and higher-order **fix** can input another session type. However we focus on predicative quantification in this paper.

## 8 Related Works

[25] is the only other formalization of dependent session types (in the same sense as ours) to our best knowledge. It is based on intuitionistic linear type theory for a variant of  $\pi$ -calculus, which extends the work in [2] where a kind of Curry-Howard isomorphism is established

between propositions in intuitionistic linear logic and session types for  $\pi$ -calculus. [25] concerns with two layers, an unspecified dependently typed layer for functional terms that assign meanings to atomic propositions, and a session typed layer that composes sessions and interprets linear logic connectives. Quantifiers connect these two layers where universal quantifier inputs a functional term and existential quantifier outputs a functional term. Cut of  $\forall/\exists$  is communication. Their line of works presents session types in a polarized style, corresponding to their left/right introduction/elimination rules of the logic. Our work is different in many ways. Our work is based on  $\lambda$ -calculus instead of  $\pi$ -calculus/linear logic, and we have shown our concrete implementation to support the argument that such formulation is practical to implement. Quantifiers are handled slightly differently. We present unpolarized global quantifiers in the session type, then locally interpreted it as  $\forall/\exists$  through our session API. However, the input/output action is not limited to follow the quantifiers immediately as they do. Our unpolarized style is easier to extend to multi-party sessions, while theirs is inherently binary due to the nature of duality in linear logic. [1] and [18] are based on [25] which focus on polymorphic session types and proof-carrying code in session types, respectively.

There are many attempts to integrate session types into practical programming languages. [19, 12, 20] embed session types into Haskell, [21] in Scala, [10] in Rust, [16] in C, and [9, 17, 8] in Java. The single salient feature is that we support dependent session types while none of above supports. Our type system also guarantees linearity and duality natively and statically without any special encoding. Due to the lack of linear types, [12] relies on an encoding of linear  $\lambda$ -calculus, [19, 20] rely on indexed monads. [10] makes use of affine types in Rust that guarantees “at most once” usage which is still not enough. Other works did not capture linearity in the type system. Duality is encoded as a proof system using type classes in [19, 12], and using traits in [10]. [21] uses Scala’s `In[-]/Out[-]` types where `-` is a *local* type, and similarly [20] uses `dual/notDual`, and they are both similar to our prior work using `chanpos` and `channeg`. [9] ensures duality in the runtime and [17, 8] are its extensions.

There are other works that are loosely related to ours, such as those investigating links between logics and session types [27, 26, 2]. Please refer to [34] for more.

## 9 Conclusion

We have presented a dependent session type system  $\mathcal{L}_{\forall, \exists}^{\pi}$  based on multi-threaded  $\lambda$ -calculus with linear types. Our type system handles quantification over static terms in session types, allowing more precise session protocols to be described elegantly. We use an unpolarized presentation that treats quantifiers in session types as global and interprets them locally as either universal quantifier for inputs or existential quantifier for outputs. Linearity is guaranteed statically by the type system, duality is guaranteed by a combination of global session types, roles at a local endpoint, and guards in the session API.  $\mathcal{L}_{\forall, \exists}^{\pi}$  also supports delegations, high-order sessions, polymorphic sessions, and recursive sessions. Our type system enjoys subject reduction and progress properties, which implies session fidelity and deadlock-freeness. We have shown the practicality of  $\mathcal{L}_{\forall, \exists}^{\pi}$  by providing a concrete back-end in Erlang/Elixir, that is asynchronous and distributed. Our formulation also bears extensions in mind and can be easily adapted to multi-party sessions based on multirole logic [35]. We will leave this as a future work.

## A Appendix - More Examples

► **Example 9** (Queue). The example comes from SILL<sup>4</sup>, an implementation of binary session types based on [2]. As compared to a simple queue, we define a dependently typed queue indexed by its length as follows, with the higher-order ‘fix’ introduced in Section 7,

$$\text{queue}(a:\text{vtype}, n:\text{int}) ::= \text{branch}(1, \text{msg}(1, a) :: \text{queue}(n+1), \\ \text{ite}(n > 0, \text{msg}(0, a) :: \text{queue}(n-1), \text{end}(0)))$$

where the client can choose to either enqueue, or dequeue an element of  $a$ . In the dequeue case, instead of encoding an optional value as a **branch** to deal with dequeuing from an empty queue, we use the length of the queue to decide the continuation of the session type. If the length  $n$  is greater than 0, the endpoint allows dequeuing. Otherwise, the endpoint can only be closed. As mentioned before, **itet/itef** are proof functions that have no runtime cost, while a non-dependently session typed queue will require **choose/offer** that need to communicate a tag at runtime. We follow their example, and present the **elem** function as follow,

```
fun elem {a:vtype} {n:nat}
  (q:chan(1,queue(a,n)), e:a): chan(1,queue(a,n+1)): void =
  let fun server {n:nat}
    (out:chan(0,queue(a,n+1)), inp:chan(C,queue(a,n))): void =
    let prval () = recurse out
    val c = offer out
    in case c of
      | Right () => let prval () = itet out
                    val () = send (out, e)
                    in cut (out, inp) end
      | Left () => let val y = recv out
                    prval () = recurse inp
                    val () = choose (inp, Left())
                    val () = send (inp, y)
                    in server (out, inp) end
    end
  in create (lam out => server (out, queue)) end
```

► **Example 10** (Polymorphism). We define a polymorphic cloud service that, given any unlimited function, will provide a service of such function. The example is taken from [1] that makes use of impredicative quantification over session types.

$$\text{service}(\pi:\text{stype}) ::= \text{branch}(1, \text{msg}(0, \text{chan}(1, \pi)) :: \text{service}, \text{end}(1)) \\ \text{cloud} ::= \text{quan}(1, \lambda\pi:\text{stype}. \text{msg}(1, \text{chan}(0, \pi) \rightarrow 1) :: \text{service}(\pi))$$

Here, **service**( $\pi$ ) is a session type scheme that says a client can repeatedly choose to use a service through a new endpoint deciplined by session type  $\pi$ , or to close it. **cloud** is a session type that says, as long as the client sends an *unlimited/non-linear* function that can provide the functionality described by  $\pi$ , the server will turn it into a replicated service.

Corresponding server and client programs could be written like the followings.

<sup>4</sup> <https://github.com/ISANobody/sill>



## 23:16 Dependent Session Types

```
implement server (ch:chan(0,cloud)): void = let
  prval () = exify ch
  val f = recv ch
  fun loop {p:stype} (ch:chan(0,service(p)), f:chan(0,p)->void): void =
    let prval () = recurse ch
      val c = offer ch
    in case c of
      | Right () => wait ch
      | Left () => let val ep = create (lam ch => f ch)
        val () = send (ch, ep)
        in loop (ch, f) end
    end
in
  loop (ch, f)
end

implement client (ch:chan(1,cloud)): void = let
  (* This is an instance of the service that does printing *)
  fun echo (ch:chan(0,msg(1,string)::end(1))): void =
    let val () = print (recv ch)
    in wait ch end

  prval () = unify ch
  val () = send (ch, echo)
  fun loop (ch:chan(1,service(msg(1,string)::end(1))), n:int): void =
    let prval () = recurse ch
    in if n <= 0
      then (choose (ch, Right()); close ch)
      else let val () = choose (ch, Left())
        val ep = recv ch
        val () = send (ep, "hello world!")
        val () = close ep
        in loop (ch, n-1) end
    end
in
  loop (ch, 10)
end
```

## B

 Appendix - Figures

■ **Figure 9** Definition of  $\rho(\cdot)$  in  $\mathcal{L}_0$

$$\begin{array}{ll}
 \rho(\mathbf{fst}(e)) &= \rho(e) & \rho(x) &= \emptyset \\
 \rho(\mathbf{snd}(e)) &= \rho(e) & \rho(dcr) &= \{dcr\} \\
 \rho(\mathbf{lam } x.e) &= \rho(e) & \rho(dcx(e_1, \dots, e_n)) &= \rho(e_1) \uplus \dots \uplus \rho(e_n) \\
 \rho(\mathbf{app}(e_1, e_2)) &= \rho(e_1) \uplus \rho(e_2) & \rho(\langle e_1, e_2 \rangle) &= \rho(e_1) \uplus \rho(e_2) \\
 \rho(\mathbf{let } \langle x_1, x_2 \rangle = e_1 \mathbf{ in } e_2) &= \rho(e_1) \uplus \rho(e_2) & \rho(\langle \rangle) &= \emptyset \\
 \rho(\mathbf{if } e \mathbf{ then } e_1 \mathbf{ else } e_2) &= \rho(e) \uplus \rho(e_1) \\
 \rho(\Pi) &= \biguplus_t \rho(\Pi(t)) \quad t \in \mathbf{dom}(\Pi) \\
 \rho(\theta) &= \biguplus_x \rho(\theta(x)) \quad x \in \mathbf{dom}(\theta)
 \end{array}$$

■ **Figure 10** Typing Rules of Multi-threaded  $\lambda$ -calculus with Linear Types

$$\begin{array}{c}
 \frac{S \models dcr : \hat{\delta}}{\Gamma; \emptyset \vdash dcr : \hat{\delta}} \mathbf{ty-res} \quad \frac{S \models dcx : (\hat{\tau}_1, \dots, \hat{\tau}_n) \Rightarrow \hat{\tau} \quad \Gamma; \Delta_i \vdash e_i : \hat{\tau}_i \quad 1 \leq i \leq n}{\Gamma; \Delta_1, \dots, \Delta_n \vdash dcx(e_1, \dots, e_n) : \hat{\tau}} \mathbf{ty-cst} \\
 \\
 \frac{}{\Gamma, x : \tau; \emptyset \vdash x : \tau} \mathbf{ty-var-i} \quad \frac{}{\Gamma; \Delta, x : \hat{\tau} \vdash x : \hat{\tau}} \mathbf{ty-var-l} \quad \frac{}{\Gamma; \emptyset \vdash \langle \rangle : \mathbf{1}} \mathbf{ty-unit} \\
 \\
 \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \quad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \mathbf{ty-tup-i} \quad \frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \quad \Gamma; \Delta_2 \vdash e_2 : \hat{\tau}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : \hat{\tau}_1 \otimes \hat{\tau}_2} \mathbf{ty-tup-l} \\
 \\
 \frac{\Gamma; \Delta \vdash e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash \mathbf{fst}(e) : \tau_1} \mathbf{ty-fst} \quad \frac{\Gamma; \Delta \vdash e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash \mathbf{snd}(e) : \tau_2} \mathbf{ty-snd} \\
 \\
 \frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \otimes \hat{\tau}_2 \quad \Gamma; \Delta_2, x_1 : \hat{\tau}_1, x_2 : \hat{\tau}_2 \vdash e_2 : \hat{\tau}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let } \langle x_1, x_2 \rangle = e_1 \mathbf{ in } e_2 : \hat{\tau}} \mathbf{ty-tup-elim} \\
 \\
 \frac{(\Gamma; \emptyset), x : \hat{\tau}_1 \vdash e : \hat{\tau}_2 \quad \rho(e) = \emptyset}{\Gamma; \emptyset \vdash \mathbf{lam } x.e : \hat{\tau}_1 \rightarrow \hat{\tau}_2} \mathbf{ty-lam-i} \quad \frac{(\Gamma; \Delta), x : \hat{\tau}_1 \vdash e : \hat{\tau}_2}{\Gamma; \Delta \vdash \mathbf{lam } x.e : \hat{\tau}_1 \multimap \hat{\tau}_2} \mathbf{ty-lam-l} \\
 \\
 \frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \rightarrow \hat{\tau}_2 \quad \Gamma; \Delta_2 \vdash e_2 : \hat{\tau}_1}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{app}(e_1, e_2) : \hat{\tau}_2} \mathbf{ty-app-i} \quad \frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \multimap \hat{\tau}_2 \quad \Gamma; \Delta_2 \vdash e_2 : \hat{\tau}_1}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{app}(e_1, e_2) : \hat{\tau}_2} \mathbf{ty-app-l} \\
 \\
 \frac{\Gamma; \Delta_1 \vdash e : \mathbf{bool} \quad \Gamma; \Delta_2 \vdash e_1 : \hat{\tau} \quad \Gamma; \Delta_2 \vdash e_2 : \hat{\tau} \quad \rho(e_1) = \rho(e_2)}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{if } e \mathbf{ then } e_1 \mathbf{ else } e_2 : \hat{\tau}} \mathbf{ty-if} \\
 \\
 \frac{\emptyset; \emptyset \vdash \Pi(0) : \hat{\tau} \quad \emptyset; \emptyset \vdash \Pi(t) : \mathbf{1} \text{ for each } t \in \mathbf{dom}(\Pi) \setminus \{0\}}{\emptyset; \emptyset \vdash \Pi : \hat{\tau}} \mathbf{ty-pool}
 \end{array}$$

■ **Figure 11** Reductions, Part A

To distinguish linear channels, we assign a natural number  $i$  to each channel as an identifier. We use  $ch$  to range over linear channels,  $ch_i$  for a channel with identifier  $i$ , and  $ch_{i,r_1}/ch_{i,r_2}$  for its dual endpoints of role  $r_1/r_2$ , respectively. Assuming  $i$  is some channel identifier and  $r_1, r_2$  are two different roles.

$$\begin{array}{c}
\frac{\Pi(t) = E[\mathbf{create}(\mathbf{lam} \ x.e)]}{\Pi \rightarrow \Pi[t := E[ch_{i,r_2}]] [t' \mapsto \mathbf{app}(\mathbf{lam} \ x.e, ch_{i,r_1})]} \mathbf{pr}\text{-create} \\
\\
\frac{\Pi(t_1) = E[\mathbf{close}(ch_{i,r_1})] \quad \Pi(t_2) = E[\mathbf{wait}(ch_{i,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[\langle \rangle]] [t_2 := E[\langle \rangle]]} \mathbf{pr}\text{-end} \\
\\
\frac{\Pi(t_1) = E[\mathbf{send}(ch_{i,r_1}, v)] \quad \Pi(t_2) = E[\mathbf{recv}(ch_{i,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[ch_{i,r_1}]] [t_2 := E[\langle v, ch_{i,r_2} \rangle]]} \mathbf{pr}\text{-msg} \\
\\
\frac{\Pi(t_1) = E[\mathbf{choosel}(ch_{i,r_1})] \quad \Pi(t_2) = E[\mathbf{offer}(ch_{i,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[ch_{i,r_1}]] [t_2 := E[\langle \top, ch_{i,r_2} \rangle]]} \mathbf{pr}\text{-branch-l} \\
\\
\frac{\Pi(t_1) = E[\mathbf{chooser}(ch_{i,r_1})] \quad \Pi(t_2) = E[\mathbf{offer}(ch_{i,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[ch_{i,r_1}]] [t_2 := E[\langle \perp, ch_{i,r_2} \rangle]]} \mathbf{pr}\text{-branch-r}
\end{array}$$

■ **Figure 12** Reductions, Part B

Let  $e$  be  $\mathbf{cut}(ch_{i,r_2}, ch_{j,r_1})$

$$\begin{array}{c}
\frac{\Pi(t_1) = E[\mathbf{close}(ch_{i,r_1})] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\mathbf{wait}(ch_{j,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[\langle \rangle]] [t := E[\langle \rangle]] [t_2 := E[\langle \rangle]]} \mathbf{pr}\text{-cut-end} \\
\\
\frac{\Pi(t_1) = E[\mathbf{send}(ch_{i,r_1}, v)] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\mathbf{recv}(ch_{j,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[ch_{i,r_1}]] [t := E[e]] [t_2 := E[\langle v, ch_{j,r_2} \rangle]]} \mathbf{pr}\text{-cut-msg} \\
\\
\frac{\Pi(t_1) = E[\mathbf{choosel}(ch_{i,r_1})] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\mathbf{offer}(ch_{j,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[ch_{i,r_1}]] [t := E[e]] [t_2 := E[\langle \top, ch_{j,r_2} \rangle]]} \mathbf{pr}\text{-cut-branch-l} \\
\\
\frac{\Pi(t_1) = E[\mathbf{chooser}(ch_{i,r_1})] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\mathbf{offer}(ch_{j,r_2})]}{\Pi \rightarrow \Pi[t_1 := E[ch_{i,r_1}]] [t := E[e]] [t_2 := E[\langle \perp, ch_{j,r_2} \rangle]]} \mathbf{pr}\text{-cut-branch-r}
\end{array}$$

---

References

---

- 1 Luís Caires, Jorge A Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral Polymorphism and Parametricity in Session-Based Communication. *ESOP*, 7792(Chapter 19):330–349, 2013.
- 2 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In *CONCUR*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 3 M Carbone, S Lindley, F Montesi, and C Schürmann. Coherence Generalises Duality: a logical explanation of multiparty session types. In *CONCUR*, pages 33:1–33:14, 2016.
- 4 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty Session Types as Coherence Proofs. *CONCUR*, pages 412–426, 2015.
- 5 Kohei Honda. Types for Dyadic Interaction. *CONCUR*, 1993.
- 6 Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer Berlin Heidelberg, Berlin, Heidelberg, March 1998.
- 7 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *POPL*, pages 273–284, 2008.
- 8 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. *ECOOP*, pages 329–353, 2010.
- 9 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. *ECOOP*, 2008.
- 10 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. *WGP@ICFP*, 2015.
- 11 Sam Lindley and J Garrett Morris. A Semantics for Propositions as Sessions. *ESOP*, 9032(Chapter 23):560–584, 2015.
- 12 Sam Lindley and J Garrett Morris. Embedding session types in Haskell. *Haskell*, pages 133–145, 2016.
- 13 Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 1992.
- 14 Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Inf. Comput.*, 100(1):41–77, 1992.
- 15 Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session Typing for a Featherweight Erlang. *COORDINATION*, 6721(Chapter 7):95–109, 2011.
- 16 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. *TOOLS*, 7304(Chapter 15):202–218, 2012.
- 17 Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryptis. Safe Parallel Programming with Session Java. *COORDINATION*, pages 110–126, 2011.
- 18 Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-Carrying Code in a Session-Typed Process Calculus. *CPP*, 2011.
- 19 Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. *Haskell*, pages 25–36, 2008.
- 20 M Sackman and S Eisenbach. Session Types in Haskell. 2008.
- 21 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. *ECOOP*, 2016.
- 22 R Shi. *Types for safe resource sharing in sequential and concurrent programming*. PhD thesis, Boston University, 2008.
- 23 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Sci. Comput. Program.*, 2013.
- 24 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. *PARLE*, 817(Chapter 34):398–413, 1994.

- 25   Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. *PPDP*, 2011.
- 26   P Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.
- 27   Philip Wadler. Propositions as sessions. *ICFP*, pages 273–286, 2012.
- 28   H Xi, D Zhu, and Y Li. Applied type system with stateful views. 2004.
- 29   Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1998.
- 30   Hongwei Xi. Applied Type System - Extended Abstract. *TYPES*, 2003.
- 31   Hongwei Xi. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.*, 2007.
- 32   Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. *POPL*, 2003.
- 33   Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. *POPL*, 1999.
- 34   Hongwei Xi, Zhiqiang Ren, Hanwen Wu, and William Blair. Session Types in a Linearly Typed Multi-Threaded Lambda-Calculus. *CoRR*, 2016.
- 35   Hongwei Xi and Hanwen Wu. Multirole Logic (Extended Abstract). *CoRR*, 2017.