Dependent Session Types

Hanwen Wu[1] **and Hongwei Xi**[2]

1 **Boston University**
`hwwu@cs.bu.edu`
2 **Boston University**
`hwxi@cs.bu.edu`

Abstract

Session types offer a type-based discipline for enforcing communication protocols in distributed programming. We have previously formalized simple session types in the setting of multi-threaded λ-calculus with linear types. In this work, we build upon our earlier work by presenting a form of dependent session types (of DML-style). The type system we formulate provides linearity and duality guarantees with no need for any runtime checks or special encodings. Our formulation of dependent session types is the first of its kind, and it is particularly suitable for practical implementation. As an example, we describe one implementation written in ATS that compiles to an Erlang/Elixir back-end.

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

1 Introduction

A session is a sequence of interactions among concurrently running programs. We assign session types [6, 7, 25, 8] to communication channels to ensure session fidelity, which means each participant in the session communicates according to a chosen protocol. Recent works [29, 28, 2, 4, 3] have established a form of Curry-Howard correspondence where logical propositions are interpreted as session types for terms in variants of π-calculus [14, 15].

Instead of π-calculus, it is also possible to formulate session types in the setting of λ-calculus [12, 36]. This paper formulates a form of dependent session types by extending our prior work [36].

More specifically, the formulation is based on Applied Type Systems ($\mathcal{ATS}$ [32, 30]), a type system supporting dependent types (of DML-style [35]), 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 separate 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.

When compared to other similar works (e.g. [26]), a very important difference of our formulation 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 channel *locally*. This is especially so when quantifiers are involved.

Suppose that we want to provide an equality testing service, which receives two integers m and n, and then sends out a boolean value indicating whether they are equal. Let us use *roles* 0 (server) and 1 (client), to refer to the two endpoints of a channel. We may use `S` for 0

© 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. Acces; Article No. 23; pp. 23:1–23:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and C for 1. We use equal for the following (static) term which describes the protocol for the equality testing service,

$$\texttt{equal} ::= \texttt{msg}(\texttt{C}, \mathbf{int}) :: \texttt{msg}(\texttt{C}, \mathbf{int}) :: \texttt{msg}(\texttt{S}, \mathbf{bool}) :: \texttt{end}(\texttt{S})$$

We use $\texttt{msg}(r, \hat{\tau})$ to mean that the endpoint r (more precisely, the party holding endpoint r) is to send a (linear) value of type $\hat{\tau}$, and :: for chaining, and $\texttt{end}(r)$ to mean that the endpoint r is to initiate the termination of the session (while the other side waits for it). With dependent session types, equal can be given a more precise definition as follows,

$$\texttt{equal} ::= \texttt{quan}(\texttt{C}, \lambda m{:}\mathit{int}.\texttt{quan}(\texttt{C}, \lambda n{:}\mathit{int}. \\ \texttt{msg}(\texttt{C}, \mathbf{int}(m)) :: \texttt{msg}(\texttt{C}, \mathbf{int}(n)) :: \texttt{msg}(\texttt{S}, \mathbf{bool}(m = n)) :: \texttt{end}(\texttt{S})))$$

where quan is a *global* encoding of quantifiers. For any role r, $\texttt{quan}(r, \cdot)$ means universal quantification at endpoint r, and dually, existential quantification at the other endpoint $(1 - r)$. In equal, quan means universal quantification at the client side, meaning the client process can send any integers onto the endpoint. Dually, quan refers to existential quantification at the server side, indicating that the server process can only send back a boolean value representing the equality of the two *received* integers. Note that **int** and **bool** are type constructors (static functions of c-sort $\mathit{int} \Rightarrow \mathit{type}$ and $\mathit{bool} \Rightarrow \mathit{type}$, respectively) while *int* and *bool* are *sorts* for static terms. Both $\mathbf{int}(i)$ and $\mathbf{bool}(b)$ are singleton types representing values that equal i and b, respectively. In ATS, which uses ML-like syntax, an example program of the type $\mathbf{chan}(\texttt{S}, \texttt{equal}) \to \mathbf{1}$ that provides such service on the server side endpoint can be written as follows,

```
fun eq_test (ch:chan(S,equal)): void = let
    prval () = exify ch (* prval denotes a proof value, that will be *)
    prval () = exify ch (* erased after type-checking                *)
      val  m = recv  ch
      val  n = recv  ch
      val () = send (ch, m = n)
in close ch end
```

Let us use this code sample to introduce some key concepts. We use (linear) *channels* for communication. A channel consists of two *endpoints*. When one process sends a value onto one endpoint, the value is automatically transmitted to the other endpoint of the channel. ch is one such endpoint of the channel at party S, whose type is $\mathbf{chan}(\texttt{S}, \texttt{equal})$. The *linear* type constructor, **chan**, will construct a linear type $\mathbf{chan}(r, \pi)$ given a role r and a global session type π. The combination of r and π is where a global session type gets "projected" locally. This can be used to type an endpoint of a channel at party r. As equal is globally quantified by session type constructor quan, we need to locally interpret it at party S, by calling a session API exify twice, which essentially turns $\mathbf{chan}(\texttt{S}, \texttt{equal})$ into

$$\exists m{:}\mathit{int}.\exists n{:}\mathit{int}.\mathbf{chan}(\texttt{S}, \texttt{msg}(\texttt{C}, \mathbf{int}(m)) :: \texttt{msg}(\texttt{C}, \mathbf{int}(n)) :: \texttt{msg}(\texttt{S}, \mathbf{bool}(m = n)) :: \texttt{end}(\texttt{S}))$$

for use with other session API, e.g. recv. The *guard* in the signature of exify (see Figure 13), $r \neq r_0$, specifies that, for any $\texttt{quan}(r_0, \cdot)$ at endpoint $\mathbf{chan}(r, \cdot)$, only when $r \neq r_0$ is true that exify can be invoked to turn $\mathbf{chan}(r, \texttt{quan}(r_0, \cdot))$ into $\exists a{:}\sigma.\mathbf{chan}(r, \cdot)$. Dually, before the client can use the channel to send two integers, it has to locally interpret quan at party C, by calling unify (see Figure 13) whose guard is $r = r_0$, which is the dual of exify since roles can only be 0 or 1 in a binary session. It will turn the endpoint at the client side into

$$\forall m{:}\mathit{int}.\forall n{:}\mathit{int}.\mathbf{chan}(\texttt{C}, \texttt{msg}(\texttt{C}, \mathbf{int}(m)) :: \texttt{msg}(\texttt{C}, \mathbf{int}(n)) :: \texttt{msg}(\texttt{S}, \mathbf{bool}(m = n)) :: \texttt{end}(\texttt{S}))$$

Essentially, a universally quantified endpoint *inputs* a static term from the user to eliminate the quantifier, 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 as mentioned above. So "inputs from the user" means the user writes a program to *send* a value using the endpoint. Such a twist is found in other works as well, e.g. [29, 28].

The main contribution of this paper lies in the formulation of a form of dependent session types (of DML-style) in the setting of λ-calculus, which is the first of its kind. In particular, this formulation is based on unpolarized presentation. Our technical results include preservation and progress properties, which guarantee session fidelity and deadlock-freeness. We also mention at the end an implementation of our system that targets Erlang/Elixir.

The rest of the paper is organised as follows. Section 2 briefly sets up multi-threaded λ-calculus with linear types, denoted as $\mathcal{L}_0$. Section 3 introduces *predicatization* to extend $\mathcal{L}_0$ into multi-threaded λ-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 $\mathcal{L}^{\pi}_{\forall,\exists}$. 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 λ-calculus with Linear Types

The formulation of multi-threaded λ-calculus with linear types is largely standard and follows exactly from our previous work [36] 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

Figure 1 Syntax of Multi-threaded λ-calculus with Linear Types

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

The syntax is shown in Figure 1 which is mostly standard. $\delta/\hat{\delta}$ are non-linear/linear base types. "vtype" is just linear type. Note that a type τ is also a linear type $\hat{\tau}$, but it is not regarded as a *true* linear type. *dcc*/*dcf* are dynamic constant constructors/functions (pre-defined 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 $\vec{\cdot}$ stands for a possibly empty sequence of $\cdot$, i.e. $\vec{e}$ is a possibly empty sequence of dynamic terms. $dcx(\vec{e})$ is a term of type τ if dcx is a constant of c-type $(\tau_1, \ldots, \tau_n) \Rightarrow \tau$ in $\mathcal{S}$ and for each e_i $(1 \leq i \leq n)$ in $\vec{e}$, e_i has type τ_i.

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

Typing contexts are divided into a non-linear part Γ and a linear part Δ. They are intuitionistic meaning that it is required that each variable occurs at most once in a non-linear context Γ or a linear context Δ. Given Γ_1, Γ_2 s.t. $\mathbf{dom}(\Gamma_1) \cap \mathbf{dom}(\Gamma_2) = \varnothing$, we write (Γ_1, Γ_2) for the union of the two. The same notion also applies to linear context Δ. Given non-linear context Γ and linear context Δ, we can form a combined context $(\Gamma; \Delta)$ when $\mathbf{dom}(\Gamma) \cap \mathbf{dom}(\Delta) = \varnothing$. 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 dcx 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 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 [36] and define $\rho(\cdot)$ (Figure 8) to compute the multiset (bag) of constant resources in a given expression and $\mathcal{R}$ (**RES** in [36]) to range over such multisets of resources. We say R is valid if $R \in \mathcal{R}$ holds. Intuitively, $\mathcal{R}$ can be thought as 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 Sementics

Typing rules are the same as [36], and we push it to Figure 9 in the appendix. The c-type judgment based on the signature is of the form $\mathcal{S} \vDash e : \hat{\tau}$. A typing judgment is of the form $\Gamma; \Delta \vdash e : \hat{\tau}$ which is standard. By inspecting the rules in Figure 9, we can readily see that a closed value cannot contain resources if it can be assigned a non-linear type τ. The *Lemma of Canonical Forms* and the *Lemma of Substitution* are the same as our previous work ([36] 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 just reduction on pools and properties of $\mathcal{L}_0$. Given pools Π_1, Π_2, we define *reductions on pools* $\Pi_1 \rightarrow \Pi_2$ as follows,

$$\frac{e_1 \rightarrow e_2}{\Pi[t \mapsto e_1] \rightarrow \Pi[t \mapsto e_2]}\,\mathbf{pr0} \qquad \frac{t > 0}{\Pi[t \mapsto \langle\rangle] \rightarrow \Pi}\,\mathbf{pr2}$$

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

▶ **Theorem 1** (Subject Reduction on Pools). *Assume $\varnothing; \varnothing \vdash \Pi_1 : \hat{\tau}$ is derivable and $\Pi_1 \rightarrow \Pi_2$ holds for some Π_2 satisfying $\rho(\Pi_2) \in \mathcal{R}$. Then $\varnothing; \varnothing \vdash \Pi_2 : \hat{\tau}$ is also derivable.*

Figure 2 Syntax of Statics

base sorts	b	$::=$	$int \mid bool \mid type \mid vtype$
sorts	σ	$::=$	$b \mid \sigma_1 \to \sigma_2$
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	Σ	$::=$	$\varnothing \mid \Sigma, a : \sigma$
static signature	$\mathcal{S}$	$::=$	$\varnothing \mid \mathcal{S}, scx : (\vec{\sigma}) \Rightarrow \sigma$
static substitutions	θ	$::=$	$[] \mid \theta[a \mapsto s]$

▶ **Theorem 2** (Progress Property on Pools). *Assume that $\varnothing; \varnothing \vdash \Pi_1 : \hat{\tau}$ is derivable. Then we have*

- *Π_1 is a singleton mapping $[0 \mapsto v]$ for some value v, or*
- *$\Pi_1 \to \Pi_2$ holds for some Π_2 s.t. $\rho(\Pi_2) \in \mathcal{R}$.*

▶ **Theorem 3** (Soundness of $\mathcal{L}_0$). *Assume that $\varnothing; \varnothing \vdash \Pi_1 : \hat{\tau}$ is derivable. Then for any Π_2, $\Pi_1 \to^* \Pi_2$ implies that either Π_2 is a singleton mapping $[0 \mapsto v]$ for some value v or $\Pi_2 \to \Pi_3$ for some Π_3 satisfying $\rho(\Pi_3) \in \mathcal{R}$, where $\to^*$ is the transitive and reflective closure of $\to$.*

Proof. Follows directly from Theorem 1 and Theorem 2. ◀

3 Predicatization

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 *predicatization* and is mostly standard in the framework of $\mathcal{ATS}$ [32]. Predicatization is extensively described in [31, 35, 33], and has been employed in several other papers based on $\mathcal{ATS}$, e.g. [24, 23]. We thus only summarize the process to prepare for the development of $\mathcal{L}^{\pi}_{\forall,\exists}$, 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 predicatization 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 predicatization 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.
- 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 λ-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.

Figure 3 Types

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

Figure 4 Extended Dynamic Language Syntax

$$
\begin{array}{llcl}
\text{dynamic terms} & e & ::= & \cdots \mid \supset^+(v) \mid \supset^-(e) \mid \wedge(e) \mid \textbf{let } \wedge(x) = e_1 \textbf{ in } e_2 \mid \\
 & & & \forall^+(v) \mid \forall^-(e) \mid \exists(e) \mid \textbf{let } \exists(x) = e_1 \textbf{ in } e_2 \\
\text{dynamic values} & v & ::= & \cdots \mid \supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)
\end{array}
$$

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

$$
\frac{\Sigma, a : \sigma; \vec{P}; \Gamma; \Delta \vdash v : \hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \forall^+(v) : \forall a{:}\sigma.\hat{\tau}}\ \textbf{ty-}\forall\textbf{-intr}
\qquad
\frac{\Sigma \vdash s : \sigma \quad \Sigma; \vec{P}; \Gamma; \Delta \vdash e : \forall a{:}\sigma.\hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \forall^-(e) : \hat{\tau}[a \mapsto s]}\ \textbf{ty-}\forall\textbf{-elim}
$$

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

Given a proposition P (a static term of sort *bool*) and a type τ, $P \supset \tau$ is a *guarded type*, and $P \wedge \tau$ is an *asserting type*. Formal definition of guarded types and asserting types can be found in [33]. Intuitively, in order to turn a value of type $P \supset \tau$ into a value of type τ, 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 Σ is the 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 1, Theorem 2, and Theorem 3 can be carrier over to $\mathcal{L}_{\forall,\exists}$ following the proof in [33].

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 [33], 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. Based on $\mathcal{L}_{\forall,\exists}$, we first extend the statics, then extend the dynamics, and finally discuss the soundness of $\mathcal{L}^{\pi}_{\forall,\exists}$.

4.1 Extending Statics

The syntax of extended statics is given in Figure 6. We add *stype* as a new base sort to represnet session types. Session types π are now static terms of sort *stype*. We use i for static integers and b for static booleans. $\mathtt{end}(i)$ means party i (the party holding endpoint i) will close the session while the other party will wait for closing. Given linear type $\hat{\tau}$ and a session type π, $\mathtt{msg}(i, \hat{\tau}) :: \pi$ means party i should send a message to the other party, and then continue as π. $\mathtt{branch}(i, \pi_1, \pi_2)$ is for branching, where party i should choose to continue as π_1 or π_2 while the other party simply follows the choice. Beyond these basic session type constructs, we have ite[1] for conditional branch, quan for universal/existential quantification, and fix for recursions. Given a static boolean expression, $\mathtt{ite}(b, \pi_1, \pi_2)$ represents π_1 when b is $\top$ (true), or π_2 when b is $\bot$ (false). Given a static function of sort $\sigma \to \mathit{stype}$, $\mathtt{quan}(i, \lambda a{:}\sigma.\pi)$ is interpreted *intuitively*[2] 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 existance of such quan for every sort σ. 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 $\mathit{stype} \to \mathit{stype}$, $\mathtt{fix}(\lambda a{:}\mathit{stype}.\pi)$ is an encoding of the fixpoint operator that represents the fixpoint of the input function. In practice, we may write recursive definitions directly as a syntax sugar (as shown in Example 8).

Figure 6 Syntax of Dependent Session Types

$$
\begin{array}{rrcl}
\text{base sorts} & b & ::= & \cdots \mid \mathit{stype} \\
\text{stypes} & \pi & ::= & \mathtt{end}(i) \mid \mathtt{msg}(i, \hat{\tau}) :: \pi \mid \mathtt{branch}(i, \pi_1, \pi_2) \mid \\
 & & & \mathtt{ite}(b, \pi_1, \pi_2) \mid \mathtt{quan}(i, \lambda a{:}\sigma.\pi) \mid \mathtt{fix}(\lambda a{:}\mathit{stype}.\pi)
\end{array}
$$

Besides, we also introduce *role* as a subset sort $\{r{:}\mathit{int} \mid r = 0 \lor r = 1\}$ 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 [35]. For instance, $\forall r{:}\mathit{role}.\mathbf{int}(r)$ is desugared into $\forall r{:}\mathit{int}.(r = 0 \lor r = 1) \supset \mathbf{int}(r)$. We also add the following *linear* type constructor as a static constant[3],

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

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

4.2 Extending Dynamics

We add the following dynamic constant functions (pre-defined functions), shown in Figure 13, to create, use, and consume linear channels. We will refer to them as *session API* or just

[1] Note that branch is just a special case of ite and we can indeed encode branch using ite.
[2] This is only intuitively interpreted. Its accurate interpretation should be considered together with an endpoint since π is global. See later sections.
[3] It is indeed $\mathbf{chan} : (\mathit{int}, \mathit{stype}) \Rightarrow \mathit{vtype}$ since in $\mathcal{ATS}$, subset sort is not allowed in a c-sort. We use *role* here just to simplify our presentation.

the API. We break up the figure and present them with explanations here.

$$\texttt{create} : \forall r_1, r_2{:}\mathit{role}.\forall \pi{:}\mathit{stype}.(r_1 \neq r_2) \supset (\mathbf{chan}(r_2, \pi) \multimap \mathbf{1}) \Rightarrow \mathbf{chan}(r_1, \pi)$$

`create` is to create a session of two threads, connected via a channel of session type π, and each thread holds an endpoint of the channel. One party is holding endpoint r_1 of type $\mathbf{chan}(r_1, \pi)$ as returned by `create` in the current thread, while the other party is holding endpoint $r_2(\neq r_1)$ of type $\mathbf{chan}(r_2, \pi)$ in a newly spawned thread evaluating the given *linear* function of type $\mathbf{chan}(r_2, \pi) \multimap \mathbf{1}$. 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.

$$\texttt{send} : \forall r, r_0{:}\mathit{role}.\forall \pi{:}\mathit{stype}.\forall \hat{\tau}{:}\mathit{vtype}.(r = r_0) \supset (\mathbf{chan}(r, \texttt{msg}(r_0, \hat{\tau}) :: \pi), \hat{\tau}) \Rightarrow \mathbf{chan}(r, \pi)$$
$$\texttt{recv} : \forall r, r_0{:}\mathit{role}.\forall \pi{:}\mathit{stype}.\forall \hat{\tau}{:}\mathit{vtype}.(r \neq r_0) \supset \mathbf{chan}(r, \texttt{msg}(r_0, \hat{\tau}) :: \pi) \Rightarrow \hat{\tau} \otimes \mathbf{chan}(r, \pi)$$

`send` is for sending linear values. Given global session type $\texttt{msg}(r_0, \hat{\tau}) :: \pi$, its interpretation at r where $r = r_0$ is to send a message of linear type $\hat{\tau}$ then to proceed as π. The `send` function *consumes* the channel, uses the capability of sending denoted by $\texttt{msg}(r_0, \hat{\tau})$, and returns another channel of type $\mathbf{chan}(r, \pi)$, where the sending capability is now removed. Dually, the interpretation of $\texttt{msg}(r_0, \hat{\tau}) :: \pi$ is to receive at party $r(\neq r_0)$, implemented by `recv`. 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,

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

Similarly, `close` is for terminating a session while `wait` is waiting for the other side to close.

$$\texttt{close} : \forall r, r_0{:}\mathit{role}.(r = r_0) \supset \mathbf{chan}(r, \texttt{end}(r_0)) \Rightarrow \mathbf{1}$$
$$\texttt{wait} : \forall r, r_0{:}\mathit{role}.(r \neq r_0) \supset \mathbf{chan}(r, \texttt{end}(r_0)) \Rightarrow \mathbf{1}$$

The interpretation of $\texttt{branch}(r_0, \pi_1, \pi_2)$ at party $r(\neq r_0)$ is to offer two choices, π_1 and π_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 π_1, π_2 using the received tag b as the condition. Dually, `choose` will choose π_1 and π_2 respectively according to the boolean tag provided by the user. Note that these two functions are completely unnecessary since they can be encoded using other functions/session types. We present them here just to stay inline with others where `offer/choose` are usually treated as standard constructs.

$$\begin{aligned}\texttt{offer} : {} & \forall r, r_0{:}\mathit{role}.\forall \pi_1, \pi_2{:}\mathit{stype}.(r \neq r_0) \supset \mathbf{chan}(r, \texttt{branch}(r_0, \pi_1, \pi_2)) \\ & \Rightarrow \exists b{:}\mathit{bool}.\mathbf{bool}(b) \otimes \mathbf{chan}(r, \texttt{ite}(b, \pi_1, \pi_2))\end{aligned}$$
$$\begin{aligned}\texttt{choose} : {} & \forall r, r_0{:}\mathit{role}.\forall \pi_1, \pi_2{:}\mathit{stype}.\forall b{:}\mathit{bool}.(r = r_0) \supset (\mathbf{chan}(r, \texttt{branch}(r_0, \pi_1, \pi_2)), \mathbf{bool}(b)) \\ & \Rightarrow \mathbf{chan}(r, \texttt{ite}(b, \pi_1, \pi_2))\end{aligned}$$

`unify` is to interpret $\texttt{quan}(r_0, \cdot)$ at party $r(= r_0)$ as universal quantifier, while `exify` is to interpret it dually as existential quantifier at party $r(\neq r_0)$.

$$\begin{aligned}\texttt{unify} : \forall r, r_0{:}\mathit{role}.\forall \pi{:}\mathit{stype}.\forall f{:}\sigma \to \mathit{stype}.&\\ (r = r_0) \supset \mathbf{chan}(r, \texttt{quan}(r_0, f)) &\Rightarrow \forall s{:}\sigma.\mathbf{chan}(r, f(s))\\ \texttt{exify} : \forall r, r_0{:}\mathit{role}.\forall \pi{:}\mathit{stype}.\forall f{:}\sigma \to \mathit{stype}.&\\ (r \neq r_0) \supset \mathbf{chan}(r, \texttt{quan}(r_0, f)) &\Rightarrow \exists s{:}\sigma.\mathbf{chan}(r, f(s))\end{aligned}$$

`itet` and `itef` reduces the conditional branching session type $\texttt{ite}(b, \pi_1, \pi_2)$ according to static boolean expression b. `recurse` unrolls the fixpoint encoding.

$$\begin{aligned}\texttt{itet} &: \forall r{:}\mathit{role}.\forall \pi_1, \pi_2{:}\mathit{stype}.\mathbf{chan}(r, \texttt{ite}(\top, \pi_1, \pi_2)) \Rightarrow \mathbf{chan}(r, \pi_1)\\ \texttt{itef} &: \forall r{:}\mathit{role}.\forall \pi_1, \pi_2{:}\mathit{stype}.\mathbf{chan}(r, \texttt{ite}(\bot, \pi_1, \pi_2)) \Rightarrow \mathbf{chan}(r, \pi_2)\\ \texttt{recurse} &: \forall r{:}\mathit{role}.\forall f{:}\mathit{stype} \to \mathit{stype}.\mathbf{chan}(r, \texttt{fix}(f)) \Rightarrow \mathbf{chan}(r, f(\texttt{fix}(f)))\end{aligned}$$

Note that these functions (`unify`/`exify`/`itet`/`itef`/`recurse`) 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 [13, 20, 11]. Instead, we choose to make the duality as general as possible and use a *global* session type π 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 the duality checking problem into a simple integer comparison problem, which greatly simplifies our formulation. Also, it reduces the number of the dynamic constants in Figure 13 in half by avoiding coercion between so-called input/output types [13]. In our previous work [36], we used a polarized presentation, e.g. $\mathbf{chanpos}(p)$ and $\mathbf{channeg}(p)$ where p is a *local* type. This is similar to `In[]`/`Out[]` in [22], $S_?/S_!$ in [13] Section 6, and *dual*/*notDual* in [21]. We found this polarized presentation is not suitable for extending to multi-party sessions, whereas our "global+role+guard" formulation can be very easily adapted to multi-party sessions based on [37]. For example, in a three-party session, we can define $\mathbf{chan}(0, \pi)$, $\mathbf{chan}(1, \pi)$, and $\mathbf{chan}(2, \pi)$ to be *compatible*, as a generalization to duality. We very briefly mention such extension in Section 7.

$$\texttt{cut} : \forall r_1, r_2{:}\mathit{role}.\forall \pi{:}\mathit{stype}.(r_1 \neq r_2) \supset (\mathbf{chan}(r_1, \pi), \mathbf{chan}(r_2, \pi)) \Rightarrow \mathbf{1}$$

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. We will explain more in Section 5.

4.3 Dynamic Semantics

The dynamic semantics of $\mathcal{L}^{\pi}_{\forall,\exists}$ 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 ruls on pools in Figure 15 and Figure 16 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 Π is of the form $E[\texttt{send}(ch_{i,r_1}, v)]$, and thread t_2 in pool Π is of the form $E[\texttt{recv}(ch_{i,r_2})]$, then Π 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 1 can be easily established for $\mathcal{L}^{\pi}_{\forall,\exists}$, Theorem 2 is more involved due to the addition of session API. However, based on [32, 35], $\mathcal{L}_{\forall,\exists}$ and $\mathcal{L}^{\pi}_{\forall,\exists}$ are *conservative* extensions of $\mathcal{L}_0$, and the deadlock-freeness is proven for $\mathcal{L}_0$ with channels in [36] using a technique known as *DF-Reducibility*. Thus the same results can be proven for $\mathcal{L}^{\pi}_{\forall,\exists}$ using the exact same technique since the dynamic semantics are the same. We thus refer readers to [36, 35] for detailed proofs. We can then establish the same deadlock-freeness guarantee as stated in Lemma 3.1 of [36]

▶ **Theorem 4** (Subject Reduction of $\mathcal{L}^{\pi}_{\forall,\exists}$). *Assume that $\varnothing;\varnothing;\varnothing;\varnothing \vdash \Pi_1 : \hat{\tau}$ is derivable and $\Pi_1 \to \Pi_2$ s.t. $\rho(\Pi_2) \in \mathcal{R}$. Then $\varnothing;\varnothing;\varnothing;\varnothing \vdash \Pi_2 : \hat{\tau}$ is also derivable.*

▶ **Theorem 5** (Progress Property of $\mathcal{L}^{\pi}_{\forall,\exists}$). *Assume that $\varnothing;\varnothing;\varnothing;\varnothing \vdash \Pi_1 : \hat{\tau}$ is derivable and $\rho(v)$ contains no channel endpoins for every $v : \hat{\tau}$. Then*

- *Π_1 is a singleton mapping $[0 \mapsto v]$ for some v, or*
- *$\Pi_1 \to \Pi_2$ holds for some Π_2 s.t. $\rho(\Pi_2) \in \mathcal{R}$.*

▶ **Theorem 6** (Soundness of $\mathcal{L}^{\pi}_{\forall,\exists}$). *Assume that $\varnothing;\varnothing;\varnothing;\varnothing \vdash \Pi_1 : \hat{\tau}$ is derivable and $\rho(v)$ contains no channel endpoins for every $v : \hat{\tau}$. Then for any Π_2 satisfying $\rho(\Pi_2) \in \mathcal{R}$, $\Pi_1 \to^* \Pi_2$ implies either Π_2 is a singleton mapping $[0 \mapsto v]$ for some v, or $\Pi_2 \to \Pi_3$ for some Π_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 $\mathcal{ATS}$, 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 $\mathcal{L}^{\pi}_{\forall,\exists}$, 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 linear typing guarantees *no resources leaks*. For instance, in our Erlang/Elixir back-end, there are no process leaks related to channels.

Our session API library in ATS is (almost) a direct translation of those listed in Figure 13, 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 call-by-value, which indicates *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 [34] and pattern matching, the API formulates `offer`/`choose` in a simpler way as follows,

```
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 guarded recursive data type that essentially captures the equality on session types. Also, since it is existentially quantified, the type-checker will enforce *exhuastive* case analysis on the received choice to instantiate `s`. Note that `s` as in `>> chan(r,s)` is in the scope of quantifier `[s:stype]` even though it appears before the quantifier.

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 relies on order-preserving messages and is inherently asynchronous and distributed.

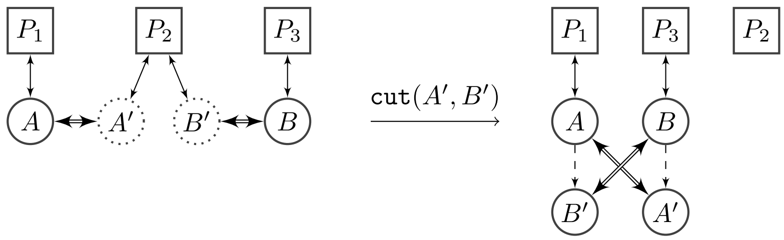
In Erlang/Elixir back-end, a message is represented by a label, a `pid`, a `ref`, and a payload. A channel endpoint is identified through a combination of a `pid` and a `ref`. The message labels are used to identify the kind of messages, e.g. `:send`/`:receive`. The `pid` is used to locate the message's 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 [16]. 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[4], 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

[4] This is because that knowing just the `pid` is enough for any process to randomly inject messages to its mailbox.

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 7 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:chan(0,p)/A':chan(1,p)` and dual endpoints `B':chan(0,p)/B: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 7, where $\leftrightarrow$ is for endpoint ownership, $\Leftrightarrow$ connects dual endpoints, and dashed arrow denotes delegation. Now, if P_1 sends a message to P_3, it will be sent through endpoint `A`, and then delivered to the mailbox of `A'`. When P_3 tries to receive the message, it will send a `:receive` request to `B`, and `B` delegates it to `A'`, and `A'` will fulfill the request since the message is in its mailbox.

We also have a shared memory implementation in ATS/C which implements our own message queue guarded by locks, and a continuation-based implementation in JavaScript using WebWorker.

6 Examples

We will show some example dependent session types or programs in the followings. We will assume that the server plays role 0 (`S`), and the client plays role 1 (`C`). We will use ATS's ML-like syntax to present the program (after omitting some insignificant details), which can be easily mapped to $\mathcal{L}^{\pi}_{\forall,\exists}$. 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 7** (Counter). One can easily define a counter as an integer stream. But more precisely, we can define dependently session typed constructor `counter` as

$$\texttt{counter}(n{:}int) ::= \texttt{branch}(\texttt{C}, \texttt{msg}(\texttt{S}, \textbf{int}(n)) :: \texttt{counter}(n+1), \texttt{end}(\texttt{C}))$$

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. `counter` makes use of higher-order fixpoint encoding, `fix`, which is better explained in Example 8. On top of `counter`, we can define a service `from` that given an integer n, returns an endpoint of session type $\texttt{counter}(n)$.

$$\texttt{from} ::= \texttt{quan}(\texttt{C}, \lambda n{:}int.\texttt{msg}(\texttt{C}, \textbf{int}(n)) :: \texttt{msg}(\texttt{S}, \textbf{chan}(\texttt{C}, \texttt{counter}(n))) :: \texttt{end}(\texttt{C}))$$

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 higher-order session type. We omit any testing code since it is similar to Example 8. Due to space limitation, we push other examples to Appendix A.

7 Extensions

We very briefly describe possible extensions of $\mathcal{L}^{\pi}_{\forall,\exists}$. First, it is straightforward to add *general recursion* to our language (not to the session type) as has been done in [36]. Second, one can always introduce a *higher-order* `fix` into session types, such as

$$\texttt{fix}(\lambda f{:}(\vec{\sigma} \to stype).\lambda \vec{a}{:}\vec{\sigma}.\pi), \vec{s})$$

where f is a static function of sort $(\vec{\sigma} \to stype) \to \vec{\sigma} \to stype$, and $\vec{s}$ are static terms of matching sorts $\vec{\sigma}$. Correspondingly, we need to introduce another `recurse` to unroll it. A higher-order `fix` will input static terms to form a new session type that dependents on these static terms. Thus these are also a form of dependent session types. Third, binary branching can be extended as well. For instance, we can introduce $\texttt{branch}(i, \pi_1, \pi_2, \pi_3), i \in \{0, 1, 2\}$ and cooresponding session API similar to `ite` to unroll it.

More importantly, we can extend $\mathcal{L}^{\pi}_{\forall,\exists}$ to support *multi-party session types* based on [37]. Roles will be extended from $\{0, 1\}$ to a larger set of natural numbers, $\textbf{chan}(r, \pi)$ will be extended to $\textbf{chan}(R, \pi)$ where R is now a *set* of roles. This is essential because of the need to represent one party's *complement* roles, which has to be a set. 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 [37].

Also, both predicative quantification (dependent types) and higher-order/impredicative quantification (polymorphism) are supported by $\mathcal{ATS}$, and our formulation naturally supports *polymorphic session types* in the sense of [1] since `quan` and higher-order `fix` can input session types to form a session type. We give such an example in Example 10. However, we focus on dependent session types in this paper.

8 Related Works

To our best knowledge, [26] is the only other formalization of dependent session types in a similar sense as ours. It is based on intuitionistic linear type theory for a variant of π-calculus, which extends the work in [2] where a kind of Curry-Howard isomorphism is established. The work 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. Their line of works presents session types in a polarized style, corresponding to the left/right introduction/elimination rules of the logic. Our work is different in many ways. Our work is based on λ-calculus instead of π-calculus/linear logic, and we have shown our concrete implementations to support the argument that such formulation is practical. 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 the logic. [1] and [19] are based on [26] which focus on polymorphic session types and proof-carrying code in session types, respectively. Our work supports polymorphic session types in the sense of [1] but we do not have space to formally address it.

There are many attempts to integrate session types into practical programming languages. [20, 13, 21] embed session types into Haskell, [22] in Scala, [11] in Rust, [17] in C, and [10, 18, 9] in Java. The single sailent feature is that we support dependent session types while none of above supports. Our type system also guarantees linearity and duality natively and staticly without any special encoding. Due to the lack of linear types, [13] relies on an encoding of linear λ-calculus, [20, 21] rely on indexed monads. [11] 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 [20, 13], and using traits in [11]. [22] uses Scala's `In[-]/Out[-]` types where `-` is a *local* type, and similarly [21] uses `dual/notDual`, and they are both similar to our prior work using **chanpos** and **channeg**. [10] ensures duality in the runtime and [18, 9] are its extensions. There are other works proposing new languages to support session types, such as [27, 5, 29] and `SILL`[5] [2], but these are not as practical in their current states.

There are other works that are loosely related to ours, such as those investigating links between logics and session types [29, 28, 2]. Please refer to [36] for more due to space limitations.

9 Conclusion

We have presented a form of dependent session type system $\mathcal{L}^{\pi}_{\forall,\exists}$ based on λ-calculus using unpolarized presentation. Our type system handles quantification over static terms in session types, allowing more precise session protocols to be described elegantly. Linearity is guaranteed statically by the type system, and duality is guaranteed by a combination of global session types, roles at a local endpoint, and guards in the session API. $\mathcal{L}^{\pi}_{\forall,\exists}$ also supports delegations, higher-order sessions, polymorphic sessions, and recursively defined sessions. Our type system enjoys subject reduction and progress properties, which guarantees session fidelity and deadlock-freeness. We have shown the practicality of $\mathcal{L}^{\pi}_{\forall,\exists}$ by providing a back-end in Erlang/Elixir, which is asynchronous, distributed, and leak-free. Our formulation can also be adapted to multi-party sessions based on multirole logic and we leave this as a future work.

[5] https://github.com/ISANobody/sill

A Appendix - More Examples

▶ **Example 8** (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 dependent session types.

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

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

$$\texttt{repeat}(\tau{:}\mathit{type}, n{:}\mathit{int}) ::= \\ \texttt{fix}(\lambda p{:}\mathit{int} \to \mathit{stype}.\lambda n{:}\mathit{int}.\texttt{ite}(n > 0, \texttt{msg}(\texttt{S}, \tau) :: p(n-1), \texttt{end}(\texttt{S})), n)$$

Note that `repeat` and `array` are session type constructors, which are just static functions returning static terms of sort *stype*. Also, the `fix` is a higher-order fixpoint described in Section 7. $\texttt{repeat}(\tau, n)$ then says, if $n > 0$ is true, the session proceeds to allow sending of a value of type τ from party `S` ($\texttt{msg}(\texttt{S}, \tau)$), then proceeds as $\texttt{repeat}(\tau, n-1)$. If $n > 0$ is false, the session can only be terminated by party `S` ($\texttt{end}(\texttt{S})$). Similarly, `array` says, party `S` is to send an integer n followed by n repeated messages described by $\texttt{repeat}(\tau, n)$. Therefore, the server side can be programmed as follows,

```
fun server {a:type} {n:nat}
    (ch:chan(S,array(a)), data:arrref(a,n), len:int(n)): void = let
    prval () = unify ch (* locally interprets the quantifier *)
      val () = send (ch, len) (* provide an instance for the quantifier *)
    fun sendarr {a:type} {n,m:nat|n<=m}
        (ch:chan(S,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 sendarr (ch, x-1, data, len) end
in sendarr (ch, len, data, len) end
```

And its type is

$$\texttt{server} : \forall \tau{:}\mathit{type}.\forall n{:}\mathit{nat}.(\mathbf{chan}(\texttt{S}, \texttt{array}(\tau)), \mathbf{arrref}(\tau, n), \mathbf{int}(n)) \to \mathbf{1}$$

where `data` is the array to be sent, whose type is indexed by the type of elements and the length of array. `len` is the length of array, whose type is a singleton integer that equals the length of `data`. `prval` denotes a proof value that has no runtime semantics. After type-checking has passed, these values will be eliminated.

▶ **Example 9** (Queue). The example comes from `SILL`[6], 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,

[6] https://github.com/ISANobody/sill

$$\texttt{queue}(\tau{:}\mathit{type}, n{:}\mathit{int}) ::= \texttt{branch}(\texttt{C}, \texttt{msg}(\texttt{C}, \tau) :: \texttt{queue}(\tau, n+1), \\ \texttt{ite}(n > 0, \texttt{msg}(\texttt{S}, \tau) :: \texttt{queue}(\tau, n-1), \texttt{end}(\texttt{S})))$$

where the client can choose to either enqueue or dequeue an element of type τ. 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 follows, which given a queue and an element `e`, constructs a new queue where `e` will be inserted into the queue as if it is the first element, and `e` will be the first to be dequeued.

```
fun elem {a:type} {n:nat}
    (q:chan(C,queue(a,n)), e:a): chan(C,queue(a,n+1)): void = let
        (* out: endpoint held by the server
         * inp: endpoint to the tail of queue
         *)
        fun server {n:nat}
            (out:chan(S,queue(a,n+1)), inp:chan(C,queue(a,n))): void =
            let prval () = recurse out (* unroll the fixpoint *)
                  val  c = offer out
             in case c of
                (* dequeue case *)
                | Right () => let prval () = itet  out
                                   val () = send (out, e)
                               (* let `inp` delegate the server *)
                               in cut (out, inp) end
                (* enqueue case *)
                |  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 the server thread, and return the client endpoint *)
        create (lam out => server (out, queue))
    end
```

▶ **Example 10** (Polymorphism)**.** We define a polymorphic cloud service that, given any unlimited function, will provide replicated services of such function. The example is taken from [1] that makes use of higher-order quantification over session types, and high-order sessions. We define polymorphic session types as follows,

$$\texttt{service}(\pi{:}\mathit{stype}) ::= \texttt{branch}(\texttt{C}, \texttt{msg}(\texttt{S}, \mathbf{chan}(\texttt{C}, \pi)) :: \texttt{service}(\pi), \texttt{end}(\texttt{C}))$$
$$\texttt{cloud} ::= \texttt{quan}(\texttt{C}, \lambda\pi{:}\mathit{stype}.\texttt{msg}(\texttt{C}, \mathbf{chan}(\texttt{S}, \pi) \to \mathbf{1}) :: \texttt{service}(\pi))$$

Here, $\texttt{service}(\pi)$ is a polymorphic session type constructor that says a client can repeatedly choose to use a service through a newly created endpoint disciplined by session type π, or to close it. `cloud` is a polymorphic session type that says, as long as the client sends an *unlimited/non-linear* function that can provide the functionality described by π, the server will turn it into a replicated service. Corresponding server and client programs could be written like the followings.

```
implement server (ch:chan(S,cloud)): void = let
    prval () = exify ch (* locally interpret `quan` as `exists` *)
      val  f = recv ch  (* receive the witness and output it to the user *)
    (* the `srv` function provides replicated services
     * by spawning a new endpoint every time the user requests
     *)
    fun srv {p:stype} (ch:chan(S,service(p)), f:chan(S,p)->void): void =
        let prval () = recurse ch
              val  c = offer ch
        in case c of
           (* the user chooses to close *)
           | Right () => wait ch
           (* the user requests one such service *)
           |  Left () => let val ep = create (lam ch => f ch)
                             val () = send (ch, ep)
                          in srv (ch, f) end
        end
in
    srv (ch, f)
end

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

    prval () = unify ch (* locally interpret `quan` as `forall` *)
      val () = send (ch, echo) (* provide an instance *)
    (* request the printing service n times *)
    fun prt (ch:chan(C,service(msg(C,string)::end(C))), n:int): void =
        let prval () = recurse ch
        in if n <= 0
           then (choose (ch, Right()); close ch)
           else let val () = choose (ch, Left())
                    (* receive the endpoint and use the service *)
                    val ep = recv ch
                    val () = send (ep, "hello world!")
                    val () = close ep
                  in prt (ch, n-1) end
        end
in
    prt (ch, 10)
end
```

B Appendix - Figures

Figure 8 Definition of $\rho(\cdot)$ in $\mathcal{L}_0$

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

Figure 9 Typing Rules of $\mathcal{L}_0$

$$
\frac{\mathcal{S} \vDash dcr : \hat{\delta}}{\Gamma; \varnothing \vdash dcr : \hat{\delta}}\ \textbf{ty-res}
\qquad
\frac{\begin{array}{c}\mathcal{S} \vDash dcx : (\hat{\tau}_1, \ldots, \hat{\tau}_n) \Rightarrow \hat{\tau} \\ \Gamma; \Delta_i \vdash e_i : \hat{\tau}_i \quad 1 \leqslant i \leqslant n\end{array}}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash dcx(e_1, \ldots, e_n) : \hat{\tau}}\ \textbf{ty-cst}
$$

$$
\frac{}{\Gamma, x : \tau; \varnothing \vdash x : \tau}\ \textbf{ty-var-i}
\qquad
\frac{}{\Gamma; \Delta, x : \hat{\tau} \vdash x : \hat{\tau}}\ \textbf{ty-var-l}
\qquad
\frac{}{\Gamma; \varnothing \vdash \langle\rangle : \mathbf{1}}\ \textbf{ty-unit}
$$

$$
\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \quad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \textbf{ty-tup-i}
\qquad
\frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \quad \Gamma; \Delta_2 \vdash e_2 : \hat{\tau}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : \hat{\tau}_1 \otimes \hat{\tau}_2}\ \textbf{ty-tup-l}
$$

$$
\frac{\Gamma; \Delta \vdash e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash \mathbf{fst}(e) : \tau_1}\ \textbf{ty-fst}
\qquad
\frac{\Gamma; \Delta \vdash e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash \mathbf{snd}(e) : \tau_2}\ \textbf{ty-snd}
$$

$$
\frac{\Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \otimes \hat{\tau}_2 \quad \Gamma; \Delta_2, x_1 : \hat{\tau}_1, x_2 : \hat{\tau}_2 \vdash e_2 : \hat{\tau}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ \langle x_1, x_2 \rangle = e_1\ \mathbf{in}\ e_2 : \hat{\tau}}\ \textbf{ty-tup-elim}
$$

$$
\frac{(\Gamma; \varnothing), x : \hat{\tau}_1 \vdash e : \hat{\tau}_2 \quad \rho(e) = \varnothing}{\Gamma; \varnothing \vdash \mathbf{lam}\ x.e : \hat{\tau}_1 \to \hat{\tau}_2}\ \textbf{ty-lam-i}
\qquad
\frac{(\Gamma; \Delta), x : \hat{\tau}_1 \vdash e : \hat{\tau}_2}{\Gamma; \Delta \vdash \mathbf{lam}\ x.e : \hat{\tau}_1 \multimap \hat{\tau}_2}\ \textbf{ty-lam-l}
$$

$$
\frac{\begin{array}{c}\Gamma; \Delta_2 \vdash e_2 : \hat{\tau}_1 \\ \Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \to \hat{\tau}_2\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{app}(e_1, e_2) : \hat{\tau}_2}\ \textbf{ty-app-i}
\qquad
\frac{\begin{array}{c}\Gamma; \Delta_2 \vdash e_2 : \hat{\tau}_1 \\ \Gamma; \Delta_1 \vdash e_1 : \hat{\tau}_1 \multimap \hat{\tau}_2\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{app}(e_1, e_2) : \hat{\tau}_2}\ \textbf{ty-app-l}
$$

$$
\frac{\Gamma; \Delta_1 \vdash e : \mathbf{bool} \quad \Gamma; \Delta_2 \vdash e_1 : \hat{\tau} \quad \Gamma; \Delta_2 \vdash e_2 : \hat{\tau} \quad \rho(e_1) = \rho(e_2)}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \hat{\tau}}\ \textbf{ty-if}
$$

$$
\frac{\varnothing; \varnothing \vdash \Pi(0) : \hat{\tau} \quad \varnothing; \varnothing \vdash \Pi(t) : \mathbf{1}\ \text{for each}\ t \in \mathbf{dom}(\Pi) \backslash \{0\}}{\varnothing; \varnothing \vdash \Pi : \hat{\tau}}\ \textbf{ty-pool}
$$

Figure 10 Some Static Constants (scc) in $\mathcal{L}_{\forall,\exists}$

$$
\begin{array}{rcl@{\qquad}rcl}
\times & : & (type, type) \Rightarrow type & \otimes & : & (vtype, vtype) \Rightarrow vtype \\
\rightarrow & : & (vtype, vtype) \Rightarrow type & \multimap & : & (vtype, vtype) \Rightarrow vtype \\
\supset & : & (bool, type) \Rightarrow type & \supset & : & (bool, vtype) \Rightarrow vtype \\
\wedge & : & (bool, type) \Rightarrow type & \wedge & : & (bool, vtype) \Rightarrow vtype \\
\forall & : & (\sigma \rightarrow type) \Rightarrow type & \forall & : & (\sigma \rightarrow vtype) \Rightarrow vtype \\
\exists & : & (\sigma \rightarrow type) \Rightarrow type & \exists & : & (\sigma \rightarrow vtype) \Rightarrow vtype \\
\mathbf{int} & : & () \Rightarrow type & \mathbf{int} & : & (int) \Rightarrow type \\
\mathbf{bool} & : & () \Rightarrow type & \mathbf{bool} & : & (bool) \Rightarrow type \\
\top & : & () \Rightarrow bool & \bot & : & () \Rightarrow bool \\
\leq_{ty} & : & (type, type) \Rightarrow bool & \leq_{ty} & : & (vtype, vtype) \Rightarrow bool \\
\mathbf{1} & : & () \Rightarrow type & & &
\end{array}
$$

Figure 11 Additional Definition of $\rho(\cdot)$ in $\mathcal{L}_{\forall,\exists}$

$$
\begin{array}{rcl@{\quad}rcl}
\rho(\supset^{+}(v)) & = & \rho(v) & \rho(\forall^{+}(v)) & = & \rho(v) \\
\rho(\supset^{-}(e)) & = & \rho(e) & \rho(\forall^{-}(e)) & = & \rho(e) \\
\rho(\wedge(e)) & = & \rho(e) & \rho(\exists(e)) & = & \rho(e) \\
\rho(\mathbf{let}\ \wedge(x) = e_1\ \mathbf{in}\ e_2) & = & \rho(e_1) \uplus \rho(e_2) & \rho(\mathbf{let}\ \exists(x) = e_1\ \mathbf{in}\ e_2) & = & \rho(e_1) \uplus \rho(e_2)
\end{array}
$$

Figure 12 Additional Typing Rules of $\mathcal{L}_{\forall,\exists}$

$$
\frac{\Sigma, a:\sigma; \vec{P}; \Gamma; \Delta \vdash v : \hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \forall^{+}(v) : \forall a{:}\sigma.\hat{\tau}}\ \textbf{ty-}\forall\textbf{-intr}
\qquad
\frac{\Sigma \vdash s : \sigma \quad \Sigma; \vec{P}; \Gamma; \Delta \vdash e : \forall a{:}\sigma.\hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \forall^{-}(e) : \hat{\tau}[a \mapsto s]}\ \textbf{ty-}\forall\textbf{-elim}
$$

$$
\frac{\Sigma \vdash s : \sigma \quad \Sigma; \vec{P}; \Gamma; \Delta \vdash e : \hat{\tau}[a \mapsto s]}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \exists(e) : \exists a{:}\sigma.\hat{\tau}}\ \textbf{ty-}\exists\textbf{-intr}
\qquad
\frac{\Sigma; \vec{P}; \Gamma; \Delta \vdash e_1 : \exists a{:}\sigma.\hat{\tau}_1 \quad \Sigma, a:\sigma; \vec{P}; (\Gamma; \Delta), x : \hat{\tau}_1 \vdash e_2 : \hat{\tau}_2}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \mathbf{let}\ \exists(x) = e_1\ \mathbf{in}\ e_2 : \hat{\tau}_2}\ \textbf{ty-}\exists\textbf{-elim}
$$

$$
\frac{\Sigma; \vec{P}, P'; \Gamma; \Delta \vdash e : \hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \supset^{+}(e) : P' \supset \hat{\tau}}\ \textbf{ty-}\supset\textbf{-intr}
\qquad
\frac{\Sigma; \vec{P}; \Gamma; \Delta \vdash e : P' \supset \hat{\tau} \quad \Sigma; \vec{P} \vdash P'}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \supset^{-}(e) : \hat{\tau}}\ \textbf{ty-}\supset\textbf{-elim}
$$

$$
\frac{\Sigma; \vec{P} \vdash P' \quad \Sigma; \vec{P}; \Gamma; \Delta \vdash e : \hat{\tau}}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \wedge(e) : P' \wedge \hat{\tau}}\ \textbf{ty-}\wedge\textbf{-intr}
\qquad
\frac{\Sigma; \vec{P}; \Gamma; \Delta \vdash e_1 : P' \wedge \hat{\tau}_1 \quad \Sigma; \vec{P}, P'; (\Gamma; \Delta), x : \hat{\tau} \vdash e_2 : \hat{\tau}_2}{\Sigma; \vec{P}; \Gamma; \Delta \vdash \mathbf{let}\ \wedge(x) = e_1\ \mathbf{in}\ e_2 : \hat{\tau}_2}\ \textbf{ty-}\wedge\textbf{-elim}
$$

$$
\frac{\Sigma; \vec{P}; \Gamma; \Delta \vdash e : \hat{\tau}_1 \quad \Sigma; \vec{P} \vdash \hat{\tau}_1 \leq_{ty} \hat{\tau}_2}{\Sigma; \vec{P}; \Gamma; \Delta \vdash e : \hat{\tau}_2}\ \textbf{ty-sub}
$$

Figure 13 Extended Dynamic Constants in $\mathcal{L}^{\pi}_{\forall,\exists}$

$$
\begin{aligned}
\texttt{create} &: \forall r_1, r_2{:}\mathit{role}.\forall\pi{:}\mathit{stype}.(r_1 \neq r_2) \supset (\mathbf{chan}(r_2,\pi) \multimap \mathbf{1}) \Rightarrow \mathbf{chan}(r_1,\pi)\\
\texttt{send} &: \forall r, r_0{:}\mathit{role}.\forall\pi{:}\mathit{stype}.\forall\hat{\tau}{:}\mathit{vtype}.\\
&\qquad (r = r_0) \supset (\mathbf{chan}(r, \texttt{msg}(r_0,\hat{\tau}) :: \pi), \hat{\tau}) \Rightarrow \mathbf{chan}(r,\pi)\\
\texttt{recv} &: \forall r, r_0{:}\mathit{role}.\forall\pi{:}\mathit{stype}.\forall\hat{\tau}{:}\mathit{vtype}.\\
&\qquad (r \neq r_0) \supset \mathbf{chan}(r, \texttt{msg}(r_0,\hat{\tau} :: \pi)) \Rightarrow \hat{\tau} \otimes \mathbf{chan}(r,\pi)\\
\texttt{close} &: \forall r, r_0{:}\mathit{role}.(r = r_0) \supset \mathbf{chan}(r, \texttt{end}(r_0)) \Rightarrow \mathbf{1}\\
\texttt{wait} &: \forall r, r_0{:}\mathit{role}.(r \neq r_0) \supset \mathbf{chan}(r, \texttt{end}(r_0)) \Rightarrow \mathbf{1}\\
\texttt{offer} &: \forall r, r_0{:}\mathit{role}.\forall\pi_1,\pi_2{:}\mathit{stype}.(r \neq r_0) \supset \mathbf{chan}(r, \texttt{branch}(r_0,\pi_1,\pi_2))\\
&\qquad \Rightarrow \exists b{:}\mathit{bool}.\mathbf{bool}(b) \otimes \mathbf{chan}(r, \texttt{ite}(b,\pi_1,\pi_2))\\
\texttt{choose} &: \forall r, r_0{:}\mathit{role}.\forall\pi_1,\pi_2{:}\mathit{stype}.\forall b{:}\mathit{bool}.(r = r_0) \supset (\mathbf{chan}(r, \texttt{branch}(r_0,\pi_1,\pi_2)), \mathbf{bool}(b))\\
&\qquad \Rightarrow \mathbf{chan}(r, \texttt{ite}(b,\pi_1,\pi_2))\\
\texttt{unify} &: \forall r, r_0{:}\mathit{role}.\forall\pi{:}\mathit{stype}.\forall f{:}\sigma \to \mathit{stype}.\\
&\qquad (r = r_0) \supset \mathbf{chan}(r, \texttt{quan}(r_0, f)) \Rightarrow \forall s{:}\sigma.\mathbf{chan}(r, f(s))\\
\texttt{exify} &: \forall r, r_0{:}\mathit{role}.\forall\pi{:}\mathit{stype}.\forall f{:}\sigma \to \mathit{stype}.\\
&\qquad (r \neq r_0) \supset \mathbf{chan}(r, \texttt{quan}(r_0, f)) \Rightarrow \exists s{:}\sigma.\mathbf{chan}(r, f(s))\\
\texttt{itet} &: \forall r{:}\mathit{role}.\forall\pi_1,\pi_2{:}\mathit{stype}.\mathbf{chan}(r, \texttt{ite}(\top,\pi_1,\pi_2)) \Rightarrow \mathbf{chan}(r,\pi_1)\\
\texttt{itef} &: \forall r{:}\mathit{role}.\forall\pi_1,\pi_2{:}\mathit{stype}.\mathbf{chan}(r, \texttt{ite}(\bot,\pi_1,\pi_2)) \Rightarrow \mathbf{chan}(r,\pi_2)\\
\texttt{recurse} &: \forall r{:}\mathit{role}.\forall f{:}\mathit{stype} \to \mathit{stype}.\mathbf{chan}(r, \texttt{fix}(f)) \Rightarrow \mathbf{chan}(r, f(\texttt{fix}(f)))\\
\texttt{cut} &: \forall r_1, r_2{:}\mathit{role}.\forall\pi{:}\mathit{stype}.(r_1 \neq r_2) \supset (\mathbf{chan}(r_1,\pi), \mathbf{chan}(r_2,\pi)) \Rightarrow \mathbf{1}
\end{aligned}
$$

Figure 14 Additional Evaluation Context for $\mathcal{L}_{\forall,\exists}$

$$
\begin{aligned}
\text{evaluation context } E ::=\ & \cdots \mid {\supset^-}(E) \mid \forall^-(E) \mid\\
& \wedge(E) \mid \mathbf{let}\ \wedge(x) = E\ \mathbf{in}\ e \mid\\
& \exists(E) \mid \mathbf{let}\ \exists(x) = E\ \mathbf{in}\ e
\end{aligned}
$$

Figure 15 Reductions on Pools in $\mathcal{L}^{\pi}_{\forall,\exists}$, 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. Assuming v is some value, b is some boolean value.

$$\frac{\Pi(t) = E[\texttt{create}(\mathbf{lam}\ x.e)]}{\Pi \to \Pi[t := E[ch_{i,r_2}]][t' \mapsto \mathbf{app}(\mathbf{lam}\ x.e, ch_{i,r_1})]}\ \textbf{pr-create}$$

$$\frac{\Pi(t_1) = E[\texttt{close}(ch_{i,r_1})] \quad \Pi(t_2) = E[\texttt{wait}(ch_{i,r_2})]}{\Pi \to \Pi[t_1 := E[\langle\rangle]][t_2 := E[\langle\rangle]]}\ \textbf{pr-end}$$

$$\frac{\Pi(t_1) = E[\texttt{send}(ch_{i,r_1}, v)] \quad \Pi(t_2) = E[\texttt{recv}(ch_{i,r_2})]}{\Pi \to \Pi[t_1 := E[ch_{i,r_1}]][t_2 := E[\langle v, ch_{i,r_2}\rangle]]}\ \textbf{pr-msg}$$

$$\frac{\Pi(t_1) = E[\texttt{choose}(ch_{i,r_1}, b)] \quad \Pi(t_2) = E[\texttt{offer}(ch_{i,r_2})]}{\Pi \to \Pi[t_1 := E[ch_{i,r_1}]][t_2 := E[\langle b, ch_{i,r_2}\rangle]]}\ \textbf{pr-branch}$$

Figure 16 Reductions on Pools in $\mathcal{L}^{\pi}_{\forall,\exists}$, Part B, `cut`

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

$$\frac{\Pi(t_1) = E[\texttt{close}(ch_{i,r_1})] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\texttt{wait}(ch_{j,r_2})]}{\Pi \to \Pi[t_1 := E[\langle\rangle]][t := E[\langle\rangle]][t_2 := E[\langle\rangle]]}\ \textbf{pr-cut-end}$$

$$\frac{\Pi(t_1) = E[\texttt{send}(ch_{i,r_1}, v)] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\texttt{recv}(ch_{j,r_2})]}{\Pi \to \Pi[t_1 := E[ch_{i,r_1}]][t := E[e]][t_2 := E[\langle v, ch_{j,r_2}\rangle]]}\ \textbf{pr-cut-msg}$$

$$\frac{\Pi(t_1) = E[\texttt{choose}(ch_{i,r_1}, b)] \quad \Pi(t) = E[e] \quad \Pi(t_2) = E[\texttt{offer}(ch_{j,r_2})]}{\Pi \to \Pi[t_1 := E[ch_{i,r_1}]][t := E[e]][t_2 := E[\langle b, ch_{j,r_2}\rangle]]}\ \textbf{pr-cut-branch}$$

References

1 Luís Caires, Jorge A Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral Polymorphism and Parametricity in Session-Based Communication. *ESOP*, 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 Simon J Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 2010.

6 Kohei Honda. Types for Dyadic Interaction. *CONCUR*, 1993.

7 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.

8 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *POPL*, pages 273–284, 2008.

9 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. *ECOOP*, pages 329–353, 2010.

10 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. *ECOOP*, 2008.

11 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. *WGP@ICFP*, 2015.

12 Sam Lindley and J Garrett Morris. A Semantics for Propositions as Sessions. *ESOP*, 9032(Chapter 23):560–584, 2015.

13 Sam Lindley and J Garrett Morris. Embedding session types in Haskell. *Haskell*, pages 133–145, 2016.

14 Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 1992.

15 Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

16 Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session Typing for a Featherweight Erlang. *COORDINATION*, 6721(Chapter 7):95–109, 2011.

17 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. *TOOLS*, 7304(Chapter 15):202–218, 2012.

18 Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. Safe Parallel Programming with Session Java. *COORDINATION*, pages 110–126, 2011.

19 Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-Carrying Code in a Session-Typed Process Calculus. *CPP*, 2011.

20 Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. *Haskell*, pages 25–36, 2008.

21 M Sackman and S Eisenbach. Session Types in Haskell. 2008.

22 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. *ECOOP*, 2016.

23 R Shi. *Types for safe resource sharing in sequential and concurrent programming.* PhD thesis, Boston University, 2008.

24 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Sci. Comput. Program.*, 2013.

25 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. *PARLE*, 817(Chapter 34):398–413, 1994.

26 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. *PPDP*, 2011.

27 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-Order Processes, Functions, and Sessions - A Monadic Integration. *ESOP*, 7792(Chapter 20):350–369, 2013.

28 P Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

29 Philip Wadler. Propositions as sessions. *ICFP*, pages 273–286, 2012.

30 H Xi, D Zhu, and Y Li. Applied type system with stateful views. 2004.

31 Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1998.

32 Hongwei Xi. Applied Type System - Extended Abstract. *TYPES*, 2003.

33 Hongwei Xi. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.*, 2007.

34 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. *POPL*, 2003.

35 Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. *POPL*, 1999.

36 Hongwei Xi, Zhiqiang Ren, Hanwen Wu, and William Blair. Session Types in a Linearly Typed Multi-Threaded Lambda-Calculus. *CoRR*, 2016.

37 Hongwei Xi and Hanwen Wu. Multirole Logic (Extended Abstract). *CoRR*, 2017.