Two Session Typing Systems for Higher-Order Mobile Processes

Dimitris Mostrous and Nobuko Yoshida
Department of Computing, Imperial College London

Abstract. This paper proposes two typing systems for session interactions in higher-order mobile processes. Session types for the HOπ-calculus capture high-level structures of communication protocols and code mobility as type abstraction, and can be used to statically check the safe and consistent process composition in communication-centric distributed software. Integration of arbitrary higher-order code mobility and sessions leads to technical difficulties in type soundness, because linear usage of session channels and completion of sessions are required in executed code. By using techniques from the linear λ-calculus, we develop a coherent and tractable session typing system for the HOπ-calculus. We also present an alternative system based on fine-grained process types. The formal comparison between the two systems offers insight on the interplay between higher-order code mobility and session types.

1 Introduction

In global computing environments, applications are executed across multiple distributed sites or devices. The use of mobile code is prominent in such environments, where several participants are synthesised by communication of not only passive values but also of runnable code: for example a service can be delegated to different participants, by sending either a channel via which it is accessible, or code that accesses it; and incoming code may transit through several devices that alter their computational behaviour or their data through interaction with it.

The Higher-Order π-calculus (HOπ-calculus) [22] is a general formalism of interaction in which two kinds of mobility, name passing and process passing, are integrated in a simple and universal form: in this model, processes can be instantiated by names and other processes, just like a piece of mobile code is instantiated with local capability after migration. This additional expressiveness inherited from the λ-calculus provides a powerful basis for describing and analysing dynamicity in global computing scenarios.

As a type-theoretic foundation for highly structured communication protocols often found in distributed applications, this paper focuses on the notion of *sessions* and their types. A session is a series of communications between two parties which form a meaningful logical unit, just like a web session between a browser and a web server when a human user interacts with an e-commerce site. Session types model such interactions as an abstract structure of typed inputs and outputs. The study of session typing systems is now wide-spread due to the need for structured communications in various scenarios in distributed computing. Starting from 1994, it has been studied for the π-calculus [4, 12–14, 18, 23, 32], ambients [11], CORBA interfaces [24], Concurrent Haskell [21], multi-threaded functional languages [16, 26, 27] and distributed [9] and multi-threaded Java [8]. At the industry level, languages with variants of session types are implemented in an operating system [10] and WC3 Choreography Web Description Language [5, 6, 29].

While many advanced session types for the π-calculus and programming languages have been studied, there exist no session typing systems for the HOπ-calculus. Incorporation of sessions into the HOπ-calculus offers a general theoretical basis for examining the interplay between two non-trivial features in communication-based programming, higher-order mobility

and session-based structured interaction. This paper establishes the first session type theory for the HOπ-calculus which can statically validate the type safety of complex distributed scenarios with code mobility. In spite of their simple type syntax, the previous literature have shown that obtaining type soundness for session types is an intricate task because of delegation of sessions [32]. In addition, in the presence of higher-order process passing, with the instantiation of names into executable code, preservation of typability becomes even more non-trivial. We provide two different solutions: one by controlling the linear use of variables for higher-order processes, which enjoys simplicity and tractability; and another by exporting channel capabilities as types of processes, which needs more annotations but has wider, more flexible typability. These two methods provide a potential type-theoretic basis of future programming idioms for dynamic code mobility and structured communications [2, 19].

The next section defines the syntax, operational semantics, and demonstrates the combined use of session types and code mobility. Section 3 defines the first typing system inspired by the linear λ-calculus [28]. Section 4 outlines an alternative typing system based on the fine-grained process types of [30, 31], and discusses the trade-offs between the two approaches. Section 5 concludes with related and future work. The omitted definitions and proofs are found in the on-line Appendix [1].

2 The Higher-Order π-Calculus with Sessions

2.1 Syntax

The syntax of the calculus is given in Fig. 1, based on the π-calculus augmented with session primitives and the call-by-value λ-calculus. A session is a series of reciprocal interactions between two parties, possibly with branching, serving as a unit of type abstraction. A session is initiated over a *shared channel* and communications belonging to a session are performed via two fresh end-point channels specific to that session, called *session channels*. The indices 0 and 1 of session channels are used to distinguish the two end points, taking a similar approach to [14, 32]. We denote $\tilde{V}$ for a potentially empty vector $V_1 \ldots V_n$. "t" and "σ" denote types which will be given later. Type annotations are often omitted.

For terms, we have prefixes for declaring session connections, $!u(x).P$ for servers and $\overline{u}(x).P$ for clients. Here the identifier u represents the public interaction point over which a session may commence. The bound variable x represents the actual channel over which the session's communications will take place. Session communications are performed using the next four primitives: the input $k(x).P$, the output $\overline{k}\langle V\rangle.P$, branching $k \rhd \{l_1 : P_1; \ldots; l_n : P_n\}$ (often written as $k \rhd \{l_i : P_i\}_{i \in I}$ with index set I) which offers alternative interaction patterns, and selection $k \lhd l.P$ which chooses an available branch. $(\nu a : \sigma)P$ restricts (and binds) a channel a to the scope of P. Similarly, $(\nu\kappa)P$ binds κ_0 and κ_1, making them private to P. Other primitives are standard. We often omit $\mathbf{0}$. The *bindings* are induced by $(\nu a : \sigma)P$, $(\nu\kappa)P$, $!u(x).P$, $\overline{u}(x).P$ and $\lambda x.P$. The derived notions of bound and free identifiers, alpha equivalence and substitution are standard. We denote $\mathsf{fv}(P)$/$\mathsf{fn}(P)$ for the set of free variables/channels, respectively. We say P is *initial* if it does not contain free variables/session channels. The difference between shared and session channels is worth illustrating. $a(x).P_1 \mid a(x).P_2 \mid \overline{a}(x).Q$ is accepted, but $\overline{\kappa_0}\langle V_1\rangle.P_1 \mid \overline{\kappa_0}\langle V_2\rangle.P_2 \mid \kappa_1(x).Q$ is not, since two senders at κ_0 appear at the same time; $\overline{\kappa_0}\langle V\rangle.\kappa_0(x).P_2 \mid \kappa_1(y).\mathbf{0}$ is also unsafe because interactions between κ_0 and κ_1 do not match.

2.2 Reduction

The single-step call-by-value reduction relation, denoted $\longrightarrow$, is a binary relation from closed terms to closed terms, defined by the rules in Fig. 2. The rules are from those of the HOπ-

(Identifiers) $u,v,w ::= x,y,z$ variables | a,b,c shared channels

$k ::= x,y,z$ variables | $\kappa_i \quad i \in \{0,1\}$ session channels

(Terms)

$P,Q,R ::= V$		value
$\mid !u(x).P$		server
$\mid \overline{u}(x).P$		client
$\mid k(x).P$		input
$\mid \overline{k}\langle V\rangle.P$		output
$\mid k \rhd \{l_1:P_1;\ldots;l_n:P_n\}$		branching
$\mid k \lhd l.P$		selection
$\mid P \mid Q$		parallel
$\mid (\nu a:\sigma)P$		restriction
$\mid (\nu\kappa)P$		restriction
$\mid PQ$		application
$\mid \mathbf{0}$		nil process

(Values)

$V,V',W ::= u,v,w$		shared identifier
$\mid k,k',k''$		linear identifier
$\mid ()$		unit
$\mid \lambda(x:t).P$		abstraction

(Abbreviations)

$\ulcorner P \urcorner \stackrel{\text{def}}{=} \lambda(x:\mathsf{unit}).P \quad (x \notin \mathsf{fv}(P))$ thunk

$run \stackrel{\text{def}}{=} \lambda x.(x())$ run

Fig. 1. Syntax

(beta) $(\lambda(x:t).P)V \longrightarrow P\{V/x\}$

(conn) $!a(x).P \mid \overline{a}(z).Q \longrightarrow !a(x).P \mid (\nu\kappa)(P\{\kappa_0/x\} \mid Q\{\kappa_1/z\}) \quad \kappa_0,\kappa_1$ fresh

(comm) $\kappa_i(x).P \mid \overline{\kappa}_j\langle V\rangle.Q \longrightarrow P\{V/x\} \mid Q \quad i \neq j$

(label) $\kappa_j \rhd \{l_1:P_1;\ldots;l_n:P_n\} \mid \kappa_i \lhd l_m.P \longrightarrow P_m \mid P \quad i \neq j,\ 1 \leq m \leq n$

(app-l) $\dfrac{P \longrightarrow P'}{PQ \longrightarrow P'Q}$ (app-r) $\dfrac{Q \longrightarrow Q'}{VQ \longrightarrow VQ'}$

(par) $\dfrac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$ (res) $\dfrac{P \longrightarrow P'}{(\nu\tilde{a}:\tilde{\sigma})(\nu\tilde{\kappa})P \longrightarrow (\nu\tilde{a}:\tilde{\sigma})(\nu\tilde{\kappa})P'}$ (str) $\dfrac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$

Fig. 2. Reduction

calculus [22, 30], but with the necessary modifications for session communications. Rule (conn) establishes a new session between server and client via shared name u; fresh κ_0 and κ_1 are instantiated, and the server stays as it is, waiting another interaction. Rule (comm) transmits values between the private session channels. Note that a session channel can be sent and received (when $V = k$), with which various protocols are expressed, allowing complex nested and private structured communications. This interaction is called *higher-order session passing* (delegation). Rule (label) selects P_m (a communication version of the case reduction in the λ-calculus). We use the standard structure rules [20] $\equiv$ such as $(\nu\kappa)P \mid Q \equiv (\nu\kappa)(P \mid Q)$ if $\kappa_{i\in\{0,1\}} \notin \mathsf{fn}(Q)$ (see Appendix [1]).

2.3 Example: Business Protocol with Code Mobility

We show a simple protocol which contains essential features by which we can demonstrate the expressivity of the code mobility and session primitives of the HOπ-calculus; it consists of a combination of session establishing, code mobility, session delegation and branching. This extends a typical collaboration pattern that appears in many web service business protocols [5, 7, 29] to code mobility.

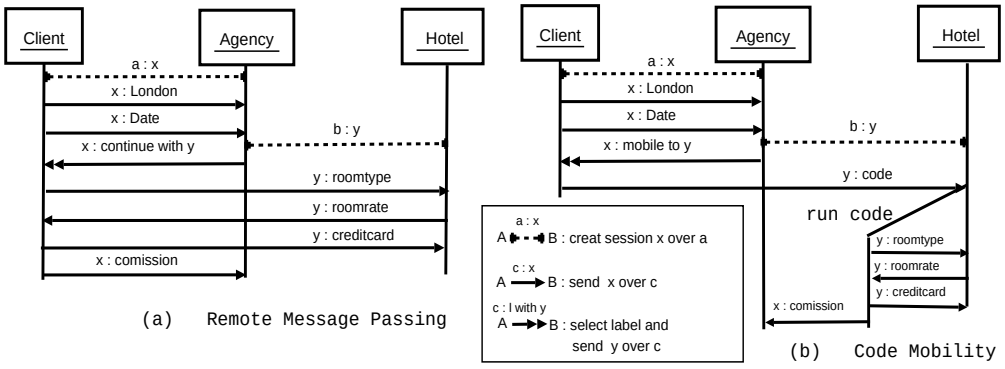

Fig. 3. Sequence Diagram for Hotel Booking

In Fig. 3, we show the sequence diagram for a protocol which models a hotel booking: first, `Booking Agency` and `Client` initiate interaction at session x over channel a; then `Client` starts exchanging a series of information with `Agency`; during this initial communication, `Agency` calculates its Round Trip Time (RTT) between `Client` and `Agency`; `Agency` selects an appropriate `Hotel` and creates a new session y over channel b with that `Hotel`. If the RTT is short (Fig. 3 (a)), then `Agency` delegates to `Client` its part of the remaining activity with `Hotel`, by sending session channel y; then `Client` and `Hotel` continue negotiations by message passing. If the RTT is long (Fig. 3 (b)), since many remote interactions increase the communication time as well as danger of communication failures, `Agency` asks back `Client` to *send mobile code* which contains the communication of the `Client`'s room plan and negotiation behaviour. `Agency` sends the code to `Hotel`, then `Hotel` runs it locally, finishing a series of interactions in its location. Finally `Agency` receives a commission fee (10 percent of the room rate) via session x, concluding the transaction.

The given scenario is straightforwardly encoded in our calculus, where session primitives make the structures of interactions clearer; we omit the subject of the intermediate communications within the same session e.g. $x \lhd l.x\langle v\rangle.x(y).P$ is written as $x \lhd l; \langle v\rangle; (y).P$. `Agency` first initiates at a and starts the interactions with `Client`; then it initiates at b and establishes session y; next it invokes either label `cont` or label `move` in `Client` depended on the RTT and sends y (higher-order session passing) to it, and waits for a completion of the transaction between `Client` and `Hotel` at x (note that "if-then-else" can be encoded using branching, and we use other base types and their operators).

$$!a(x).x(area); ...\overline{b}(y).\texttt{if rtt} < 100 \ \texttt{then}\ x \lhd \mathsf{cont}; \langle y\rangle; (z).P \quad (1)$$

$$\texttt{else}\ x \lhd \mathsf{move}; \langle y\rangle; (z).P \quad (2)$$

`Client` requests a service at a and starts a series of interaction with `Agency`, and either continues the rest of activity with `Hotel` or sends the code (a thunk in Line 4). Note that `Client` can safely send back the commission fee to `Agency` because the return message $\overline{x}\langle z \times 0.1\rangle$ which uses session channel x is embedded in the thunk.

$$\overline{a}(x).\overline{x}\langle London\rangle; ...x \rhd \{\ \mathsf{cont} : (y).y \lhd \mathsf{cont}; \langle roomtype\rangle; (z); ...\overline{x}\langle z \times 0.1\rangle\ ; \quad (3)$$

$$\mathsf{move} : (y).y \lhd \mathsf{move}; \langle \ulcorner \overline{y}\langle roomtype\rangle; (z); ...\overline{x}\langle z \times 0.1\rangle \urcorner \rangle\} \quad (4)$$

`Hotel` performs the interactions with `Agency` and `Client` via a single session at y (by the facility of higher-order session passing). In Line 6, the code sent by `Client` is run locally.

$$!b(y).y \triangleright \{\ \mathsf{cont} : (z); \langle \mathrm{roomrate}(z) \rangle; ...Q\ ; \tag{5}$$

$$\mathsf{move} : (\mathit{code}).(\mathit{run\ code} \mid y(z); \langle \mathrm{roomrate}(z) \rangle; ...Q)\} \tag{6}$$

The encoding is simple, but includes a couple of subtle points whose slight modification breaks the session structures. First, in Line 4, if we send the code which does not complete the session, then the protocol is broken: e.g. if we have interactions at y (say $\overline{y}\langle w \rangle$) after sending a thunk in Line 4 in `Client`, the session at y will appear in the three threads (two in `Hotel`, one in `Client`), so the session at y is interfered and values get mixed up. Secondly, in Line 6, if we have two or more applications (say *run code* | *run code*) instead of one *run code*, it again breaks the session structure (both at y and x). Finally, if the code is not runnable in Line 6 (like $(\lambda x.\mathbf{0})\mathit{code}$ instead of *run code*), the receiver $y(z); \langle \mathrm{roomrate}(z) \rangle; ...Q$ cannot find a matching output. Hence the variable *code* must appear exactly once and surely get instantiated into a process exactly once.

3 The First System: Higher-Order Linear Typing

3.1 Types

This section presents the first session system based on linear typing for higher-order functions. The syntax of types is given below.

$$\text{Term } \tau ::= t \mid \diamond \quad \text{Chan } \sigma ::= \mathsf{begin}.\alpha \quad \text{Val } t ::= \mathsf{unit} \mid t \to \tau \mid (t \to \tau)^1 \mid \sigma \mid \alpha$$

$$\text{Session } \alpha ::= !\lceil t \rceil.\alpha \mid ?\lceil t \rceil.\alpha \mid \oplus[l_1 : \alpha_1; \ldots; l_n : \alpha_n] \mid \&[l_1 : \alpha_1; \ldots; l_n : \alpha_n] \mid \mathsf{end}$$

It is an integration of the types from the simply typed λ-calculus with unit and the session types from the π-calculus, with the exception of linear functional types, $(t \to \tau)^1$, which represent *functions to be used exactly once*. *Term types*, ranging over τ, include all value types and the process type $\diamond$. *Channel types*, ranging over σ, take the shape $\mathsf{begin}.\alpha$. *Session types* range over $\alpha, \beta, \gamma ...$ In $\mathsf{begin}.\alpha$, begin represents the start of the session, while end represents its termination. *Value types* consist of the unit type, the function types, the linear function types and the channel and session types. Note that linear annotations are attached only to function types. In the session types, $!\lceil t \rceil.\alpha$ represents the output of a value typed by t followed by a session typed by α; $?\lceil t \rceil$ is its dual. $\oplus[l_1 : \alpha_1; \ldots; l_n : \alpha_n]$ is the selection type on which one of the labels l_i can be sent, with the subsequent session typed by α_i; $\&[l_1 : \alpha_1; \ldots; l_n : \alpha_n]$ is its dual called the branching type. We often write $\&[l_i : \alpha_i]_{i \in I}$/ $\oplus[l_i : \alpha_i]_{i \in I}$ for branching and selection types, and $\ulcorner \tau \urcorner$ for $\mathtt{unit} \to \tau$. end is often omitted. Each session type α has a *dual* type, denoted by $\overline{\alpha}$, which describes complementary behaviour. This is inductively defined as: $\overline{!\lceil t \rceil.\alpha} = ?\lceil t \rceil.\overline{\alpha}$, $\overline{\oplus[l_1 : \alpha_1; \ldots l_n : \alpha_n]} = \&[l_1 : \overline{\alpha_1}; ...; l_n : \overline{\alpha_n}]$, $\overline{?\lceil t \rceil.\alpha} = !\lceil t \rceil.\overline{\alpha}$, $\overline{\&[l_1 : \alpha_1; \ldots l_n : \alpha_n]} = \oplus[l_1 : \overline{\alpha_1}; ...; l_n : \overline{\alpha_n}]$, and $\overline{\mathsf{end}} = \mathsf{end}$.

3.2 Linear Higher-Order Typing System

We first define the two kinds of finite mappings for environments:

$$\text{Global } \Gamma ::= \emptyset \mid \Gamma, u : \sigma \mid \Gamma, x : \mathsf{unit} \mid \Gamma, x : t \to \tau \mid \Gamma, x : (t \to \tau)^1 \quad \text{Session } \Sigma ::= \emptyset \mid \Sigma, k : \alpha$$

Γ is a mapping, associating value types (*except session types*) to identifiers. Σ is a mapping from session channels to session types that records precise usage information for all free session

(Common)

(Shared) $\dfrac{t \neq (t' \to \tau)^1}{\Gamma, u\colon t; \emptyset; \emptyset \vdash u : t}$ (Session) $\dfrac{}{\Gamma; k\colon \alpha; \emptyset \vdash k : \alpha}$ (LVar) $\dfrac{}{\Gamma, x : (t \to \tau)^1; \emptyset; \{x\} \vdash x : (t \to \tau)^1}$

(Function)

(Base) $\dfrac{}{\Gamma; \emptyset; \emptyset \vdash () : \mathtt{unit}}$ (Abs) $\dfrac{\Gamma, x\colon t; \Sigma; \mathcal{S} \vdash P : \tau \ (\star)}{\Gamma; \Sigma; \mathcal{S} \setminus x \vdash \lambda(x\colon t).P : t \to \tau}$ (Abs$_S$) $\dfrac{\Gamma; \Sigma, x\colon \alpha; \mathcal{S} \vdash P : \tau}{\Gamma; \Sigma; \mathcal{S} \vdash \lambda(x\colon \alpha).P : \alpha \to \tau}$

(App) $\dfrac{\Gamma; \Sigma_1; \mathcal{S}_1 \vdash P : (t \to \tau)^1 \quad \Gamma; \Sigma_2; \mathcal{S}_2 \vdash Q : t \quad (\dagger)}{\Gamma; \Sigma_1, \Sigma_2; \mathcal{S}_1, \mathcal{S}_2 \vdash PQ : \tau}$ (Sub) $\dfrac{\Gamma; \Sigma; \mathcal{S} \vdash P : t \to \tau}{\Gamma; \Sigma; \mathcal{S} \vdash P : (t \to \tau)^1}$

(**Process**)

(Nil) $\dfrac{\Sigma = \{\tilde{k} : \tilde{\mathsf{end}}\}}{\Gamma; \Sigma; \emptyset \vdash \mathbf{0} : \diamond}$ (Par) $\dfrac{\Gamma; \Sigma_{1,2}; \mathcal{S}_{1,2} \vdash P_{1,2} : \diamond}{\Gamma; \Sigma_1, \Sigma_2; \mathcal{S}_1, \mathcal{S}_2 \vdash P_1 \mid P_2 : \diamond}$ (New) $\dfrac{\Gamma, a\colon \sigma; \Sigma; \mathcal{S} \vdash P : \diamond}{\Gamma; \Sigma; \mathcal{S} \vdash (\nu a\colon \sigma)P : \diamond}$ (Newκ) $\dfrac{\Gamma; \Sigma, \kappa_i : \alpha, \kappa_j : \overline{\alpha}; \mathcal{S} \vdash P : \diamond}{\Gamma; \Sigma; \mathcal{S} \vdash (\nu\kappa)P : \diamond}$

(Acc) $\dfrac{\Gamma; \emptyset; \emptyset \vdash u : \mathsf{begin}.\overline{\alpha} \quad \Gamma; x\colon \alpha; \emptyset \vdash P : \diamond}{\Gamma; \emptyset; \emptyset \vdash !u(x).P : \diamond}$ (Req) $\dfrac{\Gamma; \emptyset; \emptyset \vdash u : \mathsf{begin}.\alpha \quad \Gamma; \Sigma, x\colon \alpha; \mathcal{S} \vdash P : \diamond}{\Gamma; \Sigma; \mathcal{S} \vdash \overline{u}(x).P : \diamond}$

(Rec) $\dfrac{\Gamma, x\colon t; \Sigma, k\colon \alpha; \mathcal{S} \vdash P : \diamond \quad (\star)}{\Gamma; \Sigma, k\colon ?[t].\alpha; \mathcal{S} \setminus x \vdash k(x).P : \diamond}$ (Rec$_S$) $\dfrac{\Gamma; \Sigma, k\colon \alpha', x\colon \alpha; \mathcal{S} \vdash P : \diamond}{\Gamma; \Sigma, k\colon ?[\alpha].\alpha'; \mathcal{S} \vdash k(x).P : \diamond}$

(Send) $\dfrac{\Gamma; \Sigma_1; \mathcal{S}_1 \vdash P : \diamond \quad \Gamma; \Sigma_2; \mathcal{S}_2 \vdash V : t \quad k\colon \alpha \in \Sigma_{i \in \{1,2\}} \quad (\dagger)}{\Gamma; (\Sigma_1, \Sigma_2) \setminus \{k : \alpha\}, k : ![t].\alpha; \mathcal{S}_1, \mathcal{S}_2 \vdash \overline{k}\langle V \rangle.P : \diamond}$

(Bra) $\dfrac{\Gamma; \Sigma, k\colon \alpha_i; \mathcal{S} \vdash P_i : \diamond \quad (\forall i \in I)}{\Gamma; \Sigma, k : \&[l_i : \alpha_i]_{i \in I}; \mathcal{S} \vdash k \rhd \{l_i : P_i\}_{i \in I} : \diamond}$ (Sel) $\dfrac{\Gamma; \Sigma, k : \alpha_j; \mathcal{S} \vdash P : \diamond \quad j \in I}{\Gamma; \Sigma, k : \oplus[l_i : \alpha_i]_{i \in I}; \mathcal{S} \vdash k \lhd l_j.P : \diamond}$

$(\star)$ if $t = (t' \to \tau')^1$ then $x \in \mathcal{S}$. $(\dagger)$ if $t = t' \to \tau'$ then $\Sigma_2 = \mathcal{S}_2 = \emptyset$.

Fig. 4. Session Typing based on Linear Types

channels in a term, so that the cumulative result can be compared with the expected session type. In addition, we use a set of linear variables ranged over $\mathcal{S}$, $\mathcal{S}'$, ... to ensure linear usage of function terms that may contain session channels. Σ, Σ' and $\mathcal{S}, \mathcal{S}'$ denote disjoint-domain unions. $\Gamma, u\colon \sigma$ means $u \notin \mathrm{dom}(\Gamma)$. Then the typing judgement takes the shape:

$$\Gamma; \Sigma; \mathcal{S} \vdash P : \tau$$

which is read: under a global environment Γ, a term P has a type τ with session usages described by Σ and linear variables specified by $\mathcal{S}$. We say the judgement is *well-formed* if $\mathrm{dom}(\Gamma) \supseteq \mathcal{S}$ and $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Sigma) = \emptyset$. The typing system is given in Fig. 4. In each rule, we assume the environments of the consequence are defined.

In the first group, **(Common)**, (Shared) is an introduction rule for identifiers with shared types, i.e. neither $(t')^1$ or α. (Session) is for session channels and (LVar) is for linear variables, recording k in Σ and x in $\mathcal{S}$, respectively.

The second group, **(Function)**, comes from the simply typed linear λ-calculus. In (Abs), the side condition ($\star$) ensures that the formal parameter x, to be substituted with the received function, appears in the linear variables' premise. In the conclusion, we remove x from the function environment. (Abs_S) is an abstraction rule for session channels. (App) is the rule for application; the side condition (†) ensures that when the right term is of shared function type, it is required not to have free session channels or linear variables. The conclusion says that P and Q's session environments and linear variable sets are disjoint. (Sub) is a subsumption rule to lift from the shared to linear function. The converse is unsafe.

The final group, **(Process)**, are for processes integrated with linear functional typing. In (Nil), we start from the session environment only with end-usages and the empty linear variable set. In (Par), we parallel-compose two processes, assuming disjointness of session environments and linear variable sets as in (App). (New) and (Newκ) hide a shared name and a pair of session channels, respectively. The latter erases, in the session environment, complementary communication patterns for the two endpoints of κ, in order to ensure compatible dyadic interactions. (Acc) and (Req) are for initiating sessions. (Acc) forbids the use of any *free* linear identifier because of replication. The type expected for the session channel is dual ($\overline{\alpha}$) to that portion of the declared session type for the shared identifier. In (Req), it is used as it is (α). (Rec) handles the reception (input) of values. Just as (Abs), if received values have a linear function type, x should be recorded to ensure its linear usage in P. The relevant consumption is composed in the conclusion's session environment, in a way that agrees with the protocol. (Rec_S) is for the input of session channels.

(Send) is the most complex rule, integrating session typing and linear typing. Firstly, (†), as in (App), enforces safety when sending linear functions. Secondly $k\colon \alpha \in \Sigma_{i\in\{1,2\}}$ means either Σ_1 or Σ_2 contains the complete session $k\colon \alpha$ (since Σ_1, Σ_2 is defined in the conclusion). When $k\colon \alpha \in \Sigma_1$ and V has a functional type, it ensures that all occurring session channels within V being sent are complete (i.e. suffixed with end). Hence they cannot occur in the continuation P, because, if they did, we would have a race condition between the receiver of V and P, w.r.t. communications over these common channels, as noted in the example in § 2.3. This condition forces V to be k itself when it has the session type α, uniformly generalising the corresponding rule in the session types [14, 18, 23, 32]. This is important since, in the presence of higher-order mobility, the sent code containing k can be executed locally and privately in the receiver side: `Client` in the example in § 2.3 becomes typable with this general rule. In the conclusion, we delete k in either Σ_1 or Σ_2, and the relevant consumption is recorded in the conclusion's session environment. Note the function environments are disjoint. (Bra) and (Sel) are the rules for branching and selection. They are standard from [18].

3.3 Type Soundness and Type Safety

The typed processes enjoy type soundness and type safety. First, typings which start from well-formed environments construct only well-formed environments.

Proposition 3.1. *Suppose the derivation of* $\Gamma;\Sigma;\mathcal{S} \vdash P : \tau$ *starts from the axioms* (Shared, Session, LVar, Base, Nil) *of a well-formed judgement. Then* $\Gamma;\Sigma;\mathcal{S} \vdash P : \tau$ *is well-formed.*

We have the standard weakening and strengthing for Γ (but not for Σ and $\mathcal{S}$). Then the substitution lemmas follow.

Lemma 3.2 (Substitution Lemma).

1. *Suppose* $\Gamma, x\colon t;\Sigma_1;\mathcal{S}_1 \vdash P : \tau$ *and* $\Gamma;\Sigma_2;\mathcal{S}_2 \vdash V : t$ *with* $t \neq t' \to \tau'$, $x \in fv(P)$, *and* Σ_1, Σ_2 *and* $\mathcal{S}_1, \mathcal{S}_2$ *are defined. Then* $\Gamma;\Sigma_1, \Sigma_2;\mathcal{S}_1 \setminus x, \mathcal{S}_2 \vdash P\{V/x\} : \tau$.

2. *Assume* $\Gamma, x\colon t' \to \tau'; \Sigma; \mathcal{S} \vdash P : \tau$ *and* $\Gamma; \emptyset; \emptyset \vdash V : t' \to \tau'$. *Then* $\Gamma; \Sigma; \mathcal{S} \vdash P\{V/x\} : \tau$.
3. *Suppose* $\Gamma; \Sigma, x : \alpha; \mathcal{S} \vdash P : \tau$ *and* $k \notin (\mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Sigma))$. *Then* $\Gamma; \Sigma, k : \alpha; \mathcal{S} \vdash P\{k/x\} : \tau$.

Before stating the main theorems, we introduce the important notion of *balanced* session environments. Clearly, typability over arbitrary session environments is not closed under reduction. For example, the process $\overline{\kappa_0}\langle \mathtt{true} \rangle \mid \kappa_1(x).\overline{\kappa_i'}\langle x+1 \rangle$ is typable, but it reduces to $\overline{\kappa_i'}\langle \mathtt{true}+1 \rangle$, leading to a run-time error. Hence we allow only typings where the two ends of a channel are of dual types. Formally, we say that a session environment Σ is *balanced* if whenever $\kappa_i\colon \alpha, \kappa_j\colon \beta \in \Sigma$, then $\alpha = \overline{\beta}$.

Theorem 3.3 (Type Soundness).

1. *Suppose* $\Gamma; \Sigma; \mathcal{S} \vdash P : \diamond$ *with* Σ *balanced. Then* $P \equiv P'$ *implies* $\Gamma; \Sigma; \mathcal{S} \vdash P' : \diamond$.
2. *Suppose* $\Gamma; \Sigma; \mathcal{S} \vdash P : \tau$ *with* Σ *balanced. Then* $P \longrightarrow P'$ *implies* $\Gamma; \Sigma'; \mathcal{S}' \vdash P' : \tau$ *with* Σ' *balanced and* $\mathcal{S} \supseteq \mathcal{S}'$.

One may wonder why "balanced" cannot be assumed for the condition of the well-formed judgement. To see the reason, consider $(\lambda x.\overline{x}\langle 1 \rangle.(\overline{x}\langle 3 \rangle.\mathbf{0} \mid \kappa_1(x).\kappa_1(y).\mathbf{0}))\kappa_0$. This is typed under the balanced environment $\Sigma = \kappa_0 : \alpha, \kappa_1 : \overline{\alpha}$ with $\alpha = ![\mathtt{nat}].![\mathtt{nat}].\mathsf{end}$, and moreover all subterms are typed under balanced environments. This process reduces to $P' = \overline{\kappa_0}\langle 1 \rangle.(\overline{\kappa_0}\langle 3 \rangle.\mathbf{0} \mid \kappa_1(x).\kappa_1(y).\mathbf{0})$, which is still typed under the balanced environment Σ. However the body of P' (i.e. $\overline{\kappa_0}\langle 3 \rangle.\mathbf{0} \mid \kappa_1(x).\kappa_1(y).\mathbf{0}$) cannot be typed under a balanced environment. Thus if we impose the balanced condition for the typing judgement, type soundness does not hold; the general substitution lemmas (Lemma 3.2) are required for this reason, too. This is one of the subtle points on aliasing of session channels, caused by β-reductions and communications.

We now formalise type safety. First, a *k-process* is a prefixed process with subject k (such as $k(x)$ and $\overline{k}\langle V \rangle$). Next, a κ-*redex* is a pair of dual processes composed by $|$, i.e. either of forms $(\overline{\kappa_i}\langle V \rangle.P \mid \kappa_j(x).Q)$ or $(\kappa_i \lhd l_m.P \mid \kappa_j \rhd \{l_1 : Q_1; \cdots l_n : Q_n\})$ with $1 \le m \le n$. Then we say P is an *error* if $P \equiv (\nu \tilde{a})(\nu \tilde{\kappa})(Q \mid R)$ where Q is, for some κ, the $|$-composition of *either* two κ-processes that do not form a κ-redex, *or* three or more κ-processes. We then have:

Theorem 3.4 (Type Safety). *A typable process* $\Gamma; \Sigma; \mathcal{S} \vdash P : \diamond$ *with balanced* Σ *never reduces into an error.*

Typing Hotel Booking Example Using the typing system, we can now type the hotel booking example in § 2.3, guaranteeing its type safety. `Agent` has the following types at a and b.

$$\begin{array}{l} a : \mathsf{begin}.![\mathtt{string}]... \oplus [\mathsf{rtt} < 100 :\ \alpha\ ;\ \mathsf{rtt} \ge 100 : \alpha\ ],\ b : \mathsf{begin}.![\beta].\mathsf{end} \\ \quad \text{with } \alpha = \&[\mathsf{cont} : ?[\beta].![\mathtt{int}].\mathsf{end}\ ;\ \mathsf{move} : ?[\beta].![\mathtt{int}].\mathsf{end}] \\ \quad \text{and } \ \beta = \&[\mathsf{cont} :![\mathtt{string}].?[\mathtt{int}]...\mathsf{end}\ ;\ \mathsf{move} :![(\ulcorner\diamond\urcorner)^1].\mathsf{end}] \end{array}$$

Note that the type of a is dualised because a is used as the input in `Agent` (see (Acc)). α consists of higher-order session passing, and the thunk has a linear arrow type. `Client` and `Hotel` just have the dual of `Agent`'s type at a and the dual of `Agent`'s type at b, respectively. Note that in `Client`, subject y is shared in the sent code V, which is typed by (Send) with a general side condition $k : \alpha \in \Sigma_2$ explained in § 3.2.

4 The Second System: Fine-Grained Process Typing

Linear variables in the previous system "might be instantiated by a function which contains free session channels, hence it should occur exactly once": if we have *prior* knowledge as to channel capabilities with which each functional variable (hence any code instantiated into it) is associated, then we might have more flexible control over migrating code that holds session capabilities. This motivates the use of the fine grained process typing introduced in [17, 30, 31]. Consider the following server which receives thunked processes via shared channel a.

$$\mathsf{Serv}(a) \;=\; !a(x).x(y\colon \tau).\mathit{run}\ y \tag{7}$$

Since accepting arbitrary processes for execution obviously breaks access control of local resources, one might wish to restrict the behaviour of incoming code so that it can only access some specified channels. In [17, 30, 31], we introduced a type discipline which can control the effect of migrating code, by assigning a different type to each process depending on its intended use, so that a process can use a typed inputting channel (τ at a in (7) above) to detect, for example, malicious behaviour of received code via static type checking. A type for representing capability is given as a finite channel environment Δ, prescribing channel usage of each process.

$$\Gamma \vdash P \;:\; \Delta$$

This judgement means "P accesses channels at most as specified by Δ under global environment Γ". For example, under appropriate $\Gamma \supset \{b\colon \sigma, c\colon \sigma\}$ with $\sigma = \mathsf{begin}.![\mathtt{nat}].\mathsf{end}$, a client may be assigned a different type depending on its destination.

$$\Gamma \vdash \overline{b}(x).\overline{x}\langle 1\rangle :\ \{b\colon \sigma\} \quad \text{and} \quad \Gamma \vdash \overline{c}(x).\overline{x}\langle 2\rangle :\ \{c\colon \sigma\}$$

Then the following indicates a server which only accepts a process which accesses at most the specified resource, b.

$$\mathsf{Serv}(a) \;=\; !a(x).x(y\colon \ulcorner b\colon \bar{\sigma}\urcorner).\mathit{run}\ y \tag{8}$$

Using the type system in [17, 30, 31], one can check $\mathsf{Serv}(a) \mid \overline{a}(x).\overline{x}\langle \ulcorner \overline{b}(x).\overline{x}\langle 1\rangle \urcorner\rangle$ is typable while $\mathsf{Serv}(a) \mid \overline{a}(x).\overline{x}\langle \ulcorner \overline{c}(x).\overline{x}\langle 1\rangle \urcorner\rangle$ is not. Using process types with session capabilities, we can type-check that the following process is illegal:

$$k(y\colon \ulcorner k'\colon ![\mathtt{nat}]\urcorner).(\mathit{run}\ y \mid \mathit{run}\ y) \tag{9}$$

since $\mathit{run}\ y$ has a process type $\Delta = \{k'\colon ![\mathtt{nat}]\}$, and Δ and Δ are not disjoint, so two $\mathit{run}\ y$ must not be composed. Now we no longer require linear annotation on functional types. Moreover the additional type information leads to a larger typability than the previous system. For example, $k(y\colon \ulcorner k'\colon ![\mathtt{nat}]\urcorner).(\mathit{run}\ y \mid (\lambda z.\mathbf{0})y)$, $(\lambda x.\mathbf{0})\kappa_0 \mid \kappa_0(z).\mathbf{0} \mid \overline{\kappa_1}\langle 1\rangle$, and more interestingly $(\lambda x.\overline{k}\langle 1\rangle.\mathit{run}\ x)(\ulcorner \overline{k}\langle()\rangle.\mathbf{0}\urcorner)$ which do not destroy session communication but are untypable in the previous one become typable since the resulting process types are balanced.

4.1 Types

The second typing system introduced below is built on the fine-grained types of [30, 31]. The syntax of environments and types is given below.

$$\begin{array}{rl}
\text{Env} & \Gamma ::= \emptyset \mid \Gamma, x\colon t \mid \Gamma, u\colon \sigma \quad \Delta ::= \emptyset \mid \Delta, u\colon \sigma \quad \text{Term} \quad \tau ::= t \mid \Delta \\
\text{Chan} & \sigma ::= \mathsf{begin}.\alpha \mid \alpha \qquad \text{Func} \quad t ::= \mathsf{unit} \mid t \to \tau \mid \Pi x\colon \sigma.\tau \\
\text{Session} & \alpha ::= ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha \mid ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}];\alpha \mid \oplus[l_i\colon \alpha_i]_{i\in I} \mid \&[l_i\colon \alpha_i]_{i\in I} \mid \mathsf{end}
\end{array}$$

These types are from the first system except for the introduction of fines-grained process types Δ, functional dependent types $\Pi x\colon \sigma.\tau$ and channel dependent $\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}$. Note from this system, $u, v, w, \ldots$ (resp. σ) include session names and variables (resp. session types), but τ do not include channels.[1]

In $\Pi x\colon \sigma.\tau$, we allow the type τ to contain occurrences of the channel variable x; then x in τ is bound. Note $\sigma \to \tau$ is a special case of $\Pi x\colon \sigma.\tau$ with $x \notin \mathsf{fv}(\sigma)$. A process type Δ, assigned to a process, is a mapping from a finite subset of identifiers to channel types.

A channel type incorporates dependent quantification, and has the form $\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}$ indicating a vector of channels typed by $\sigma_1, .., \sigma_n$ and a vector of higher-order values typed by $t_1, .., t_m$; free occurrences of x_i in $\sigma_{i+1}, \ldots, \sigma_n$ as well as $t_1, ..., t_m$ are bound occurrences. We write $\sigma_1, ..., \sigma_n, t_1, ..., t_m$ for $\Pi(x_1\colon \sigma_1, ..., x_n\colon \sigma_n)t_1, ..., t_m$ if $x_1, ..., x_n \notin \mathsf{fv}(\sigma_1, ..., \sigma_n, t_1, ..., t_m)$. Under this abbreviation, $?[t].\beta$ is subsumed to the case $n = 0$, and $?[\alpha].\beta$ to the case $\sigma_1 = \alpha$ and $m = 0$. The set of free names and variables are defined in the standard way [30], cf. Appendix [1]. The sets of free s/variables channels incorporate those occurring in annotating types. For example, we have $\mathsf{fv}(\lambda(x\colon t).P) = (\mathsf{fv}(t) \cup \mathsf{fv}(P)) \setminus x$. Substitution by channels $P\{u/x\}$ affects not only terms but also types which annotate bound variables: for example, when the channel u is substituted for x in a process type Δ, then the types σ of x and σ' of u are joined as: $\{u_1\colon \sigma_1, ..., u_n\colon \sigma_n\}\{V/x\} = \cup_i\{u_i\{V/x\}\colon \sigma_i\{V/x\}\}$. Others are defined homomorphically. Duality is defined by adding $?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha = ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\overline{\alpha}$ and $![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha = ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\overline{\alpha}$; others remain unchanged.

4.2 Fine-Grained Process Typing System

The typing system is given in Fig. 5, and uses two kinds of judgements: the main is $\Gamma \vdash P \triangleright \Delta$, which reads "under the environment Γ, process P has an interface type Δ". Also we have $\Gamma \vdash u\colon \sigma$, which reads as "a channel u has a type σ under Γ" and the standard well-formedness $\Gamma \vdash \mathtt{Env}$ and $\Gamma \vdash \tau : \mathtt{tp}$ for environments and types following [30, 31] (which are left to Appendix [1]). For channel inference, we define the ordering $\succ$ on channel types as the smallest partial order such that: $![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha \succ \alpha$ and $\oplus[l_1\colon \alpha_1; \ldots l_n\colon \alpha_n] \succ \alpha_i$; dually for input and branching types.

The inference rules are combinations of [30] and the session typing system of the π-calculus. We use the notation $\Delta \cdot u\colon \sigma$ for $\Delta \cup \{u\colon \sigma\}$ if $\sigma = \mathsf{begin}.\alpha$; $\Delta, u\colon \sigma$ otherwise. We extend this to $\Delta \cdot \Delta'$; and $\tilde{u}\colon \tilde{\sigma}$ which means $u_1\colon \sigma_1 \cdots u_n\colon \sigma_n$.

In the first group, **(Common)**, (Val) is a standard rule for variables. (Chan) uses $\succ$ to infer *shorter* types for sessions than the type of u declared in the environment Γ. The second one, **(Function)**, comes from the simply typed λ-calculus (where the rules for the higher-order abstractions, (Abs_H) and (App_H) and those for channel abstraction, (Abs_N) and (App_N), are separately given). The final group, **(Process)**, are about processes. In (Nil), we start from the empty interface, and in (Par), we merge two interfaces together. The rule (Weak) corresponds to the process subsumption rule; since $\Gamma \vdash P : \Delta$ means "P would access channels specified at most by Δ", we can increment its interface. Note that we can*not* weaken session channels except end. (New) and (Newκ) hide a name and a pair of session channels, respectively. (Acc), (Req), (Bra) and (Sel) are standard rules for accept, request, branching and selection.

(Rec) is a combination of the input rule for session types and that in [30]. This single rule subsumes both the value input rule and the session channel input rule (recall the abbreviation

[1] For simplicity of presentation, the tail type τ does not include the channel type σ. This inclusion can be straightforwardly formalised by using the standard type equality approach [3]. We also omit the existential types from [17, 30] which are a special case of dependent types.

(Common)

(Val)
$$\frac{\Gamma, x\colon t, \Gamma' \vdash \mathtt{Env}}{\Gamma, x\colon t, \Gamma' \vdash x : t}$$

(Chan)
$$\frac{\Gamma, u\colon \sigma, \Gamma' \vdash \mathtt{Env} \quad \sigma \succ \sigma'}{\Gamma, u\colon \sigma, \Gamma' \vdash u : \sigma'}$$

(Function)

(Base)
$$\frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash () : \mathtt{unit}}$$

(Abs$_H$)
$$\frac{\Gamma, x\colon t \vdash P : \tau}{\Gamma \vdash \lambda(x{:}t).P : t \to \tau}$$

(App$_H$)
$$\frac{\Gamma \vdash P : t \to \tau \quad \Gamma \vdash Q : t}{\Gamma \vdash PQ : \tau}$$

(Abs$_N$)
$$\frac{\Gamma, x\colon \sigma \vdash P : \tau}{\Gamma \vdash \lambda(x{:}\sigma).P : \Pi x\colon \sigma.\tau}$$

(App$_N$)
$$\frac{\Gamma \vdash P : \Pi x\colon \sigma.\tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash Pu : \tau\{u/x\}}$$

(**Process**)

(Nil)
$$\frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \mathbf{0} : \emptyset}$$

(Par)
$$\frac{\Gamma \vdash P_{1,2} : \Delta_{1,2}}{\Gamma \vdash P_1 \mid P_2 : \Delta_1 \cdot \Delta_2}$$

(Weak)
$$\frac{\Gamma \vdash P : \Delta \quad \Gamma \vdash u\colon \sigma \quad \sigma \in \{\mathsf{begin}.\alpha, \mathsf{end}\} \quad u \notin \mathrm{dom}(\Delta)}{\Gamma \vdash P : \Delta, u\colon \sigma}$$

(New)
$$\frac{\Gamma, a\colon \sigma \vdash P : \Delta, a\colon \sigma}{\Gamma \vdash (\nu a\colon \sigma)P : \Delta}$$

(Newκ)
$$\frac{\Gamma, \kappa_i\colon \alpha, \kappa_j\colon \overline{\alpha} \vdash P : \Delta, \kappa_i\colon \alpha, \kappa_j\colon \overline{\alpha}}{\Gamma \vdash (\nu\kappa)P : \Delta}$$

(Acc)
$$\frac{\Gamma \vdash u : \mathsf{begin}.\overline{\alpha} \quad \{k : \beta\} \not\subseteq \Delta \quad \Gamma, x\colon \alpha \vdash P : \Delta, x\colon \alpha}{\Gamma \vdash !u(x).P : \Delta \cdot u\colon \mathsf{begin}.\alpha}$$

(Req)
$$\frac{\Gamma \vdash u : \mathsf{begin}.\alpha \quad \Gamma, x\colon \alpha \vdash P : \Delta, x\colon \alpha}{\Gamma \vdash \overline{u}(x).P : \Delta \cdot u\colon \mathsf{begin}.\alpha}$$

(Rec)
$$\frac{\Gamma \vdash k : ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}]; \alpha \quad \Gamma, \tilde{x}\colon \tilde{\sigma}, \tilde{y}\colon \tilde{t} \vdash P : \Delta, \tilde{x}\colon \tilde{\sigma}, k\colon \alpha}{\Gamma \vdash k(\tilde{x}\colon \tilde{\sigma}, \tilde{y}\colon \tilde{t}).P : \Delta, k\colon ?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}]; \alpha}$$

(Send)
$$\frac{\Gamma \vdash k : ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}]; \alpha \quad \Gamma \vdash P : \Delta \quad \{k\colon \alpha\} \subseteq \Delta \cdot \tilde{v}\colon \tilde{\sigma} \quad \Gamma \vdash V_j : t_j\{\tilde{v}/\tilde{x}\} \quad \Gamma \vdash v_i : \sigma_i\{\tilde{v}/\tilde{x}\}}{\Gamma \vdash \overline{k}\langle \tilde{v}, \tilde{V}\rangle.P : \Delta \cdot \tilde{v}\colon \tilde{\sigma} \setminus k, k\colon ![\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}]; \alpha}$$

(Bra)
$$\frac{\Gamma \vdash k : \&[l_i\colon \alpha_i]_{i\in I} \quad \Gamma \vdash P_i : \Delta, k\colon \alpha_i}{\Gamma \vdash k \rhd \{l_i : P_i\}_{i\in I} : \Delta, k\colon \&[l_i\colon \alpha_i]_{i\in I}}$$

(Sel)
$$\frac{\Gamma \vdash k : \oplus[l_i\colon \alpha_i]_{i\in I} \quad \Gamma \vdash P : \Delta, k\colon \alpha_i}{\Gamma \vdash k \lhd l_i.P : \Delta, k\colon \oplus[l_i\colon \alpha_i]_{i\in I}}$$

Fig. 5. Session Typing based on Fine-Grained Process Types

in the previous paragraph). The first assumption ensures u can input channels typed by σ_i and higher-order values typed by t_j, and in the conclusion, the free occurrences of $\tilde{x}$ in both P and t_j are bound (hence t_j is depended by $\tilde{x}$), resulting the process type Δ with a new session type $?[\Pi(\tilde{x}\colon \tilde{\sigma})\tilde{t}].\alpha$ at k (note $k \notin \mathrm{dom}(\Delta)$). (Send) is again a combination with the output rule in [30] (see also (Send) for the first system): the first assumption ensures u outputs a pair of names typed by σ_i and higher-order values typed by t_j. The third assumption says that k is either sent name v_i or a free name in P. The the first part of the arguments is v_i, then the second part of the arguments should have type $t_j\{\tilde{v}/\tilde{x}\}$ since x_i binds free occurrences of x_i in t_j. Then the effect of channel k and v_i should be recorded as a type of $\overline{k}\langle \tilde{v}, \tilde{V}\rangle$ because they will be used by the opponent input after interaction (note that we do *not* have to record the effect of V).

By essentially the same routine of the proofs in [30], we obtain:

Theorem 4.1 (Type Soundness).

1. *Suppose* $\Gamma \vdash P : t$ *and* $P \longrightarrow P'$. *Then* $\Gamma \vdash P' : t$.
2. *Suppose* $\Gamma \vdash P : \Delta$ *with* Δ *balanced. Then* $P \equiv P'$ *implies* $\Gamma \vdash P' : \Delta'$ *with* Δ' *balanced.*
3. *Suppose* $\Gamma \vdash P : \Delta$ *with* Δ *balanced. Then* $P \longrightarrow P'$ *implies* $\Gamma \vdash P' : \Delta'$ *with* Δ' *balanced.*

Note that Γ does not have to be balanced.

Theorem 4.2 (Type Safety). *A typable process* $\Gamma \vdash P : \Delta$ *with balanced* Δ *never reduces into an error.*

Typing Hotel Booking Example We revisit the hotel booking example in § 2.3. The only change from the previous types in § 3.2 is $![(\ulcorner\diamond\urcorner)^1]$ in β. This is changed to $![\Pi(x:\gamma_x, y:\gamma_y)\ulcorner\Delta\urcorner]$ with $\gamma_x = ![\mathtt{string}].?[\mathtt{int}]...\mathsf{end}$ and $\gamma_y = ![\mathtt{int}].\mathsf{end}$, and $\Delta = \{x:\gamma_x, y:\gamma_y\}$. Note that we also have to change the syntax in Line 4 from $y \lhd \mathsf{move}; \langle \ulcorner R \urcorner \rangle$ to $y \lhd \mathsf{move}; \langle x, y, \ulcorner R \urcorner \rangle$ since the type of the thunk is dependent on x and y. This suggests a trade-off between the two approaches. In the channel-dependent typing, we gain more flexibility by having more type information, but this in turn demands additional type annotation in programs. The approach based on linear typing does not need heavy annotations, though it allows the typability of a smaller, but probably pragmatically sufficient, class of programs. We may also refine the dependently typed approach with the existential types of [17, 30] (this integration is straightforward, but requires more rules), in which case we do not have to declare session names explicitly. The syntax of the example is unchanged, and the type becomes $![\exists[x:\gamma_x, y:\gamma_y]\ulcorner\Delta\urcorner]$. The reader can also check the processes in the beginning of the section are typable: in the first process, $(\lambda x.\mathbf{0})y$ has the empty process type $\emptyset$ so that we can compose with *run y* by (Par). Similarly for the second. In the third, $(\lambda(x\colon t).\overline{k}\langle 1\rangle.run\ x)(\ulcorner\overline{k}\langle()\rangle.\mathbf{0}\urcorner)$ with $t = \ulcorner k\colon ![\mathtt{unit}].\mathsf{end}\urcorner$. has a process type $\Delta = \{k\colon ![\mathtt{nat}].![\mathtt{unit}].\mathsf{end}\}$ under environment Δ. These are untypable in the first system.

4.3 Comparison of the Two Systems

We conclude this section with a comparison of the two typing systems. The examples in the beginning of this section show the existence of terms typable in the second system but not in the first system introduced in § 3. A natural question is which subsystem of the second system can precisely characterise the first, i.e. a sound and complete embedding of the first system into a subset of the second system. Observing that it is linear functions that can inhabit those types with free session capabilities (e.g. $\lambda(x\colon \alpha).\overline{x}\langle 1\rangle$ of type $\Pi x\colon \alpha.x\colon \mathtt{nat}$ is not a linear function, while $\ulcorner\overline{x}\langle 1\rangle\urcorner$ of type $\ulcorner x\colon \mathtt{nat}\urcorner$ is linear), we introduce the following three functions.

- $\mathit{Erase}(P)$ erases the dependent binding from the input and output, and $\mathit{Erase}(\tau)$ erases the dependent binding from the functional and channel dependent types; and translates process types into $\diamond$; and puts the linear annotation to a functional type which has free session typings in its tail.
- $\mathit{Proc}(\tau)$ extracts the session environment Σ from τ.
- $\mathit{Lin}(\Gamma)$ extracts the linear variable set $\mathcal{S}$ from Γ.

Formally we define:

- For terms: $\mathit{Erase}(()) = ()$, $\mathit{Erase}(u) = u$, $\mathit{Erase}(\mathbf{0}) = \mathbf{0}$, $\mathit{Erase}(k(\tilde{x}\colon \tilde{\sigma}, y\colon \tau).P) = k(y : \mathit{Erase}(\tau)).\mathit{Erase}(P)$, $\mathit{Erase}(\overline{k}\langle \tilde{v}, V\rangle.P) = \overline{k}\langle \mathit{Erase}(V)\rangle.\mathit{Erase}(P)$ and others are homomorphic. For types: $\mathit{Erase}(\mathtt{unit}) = \mathtt{unit}$, $\mathit{Erase}(\Delta) = \diamond$, $\mathit{Erase}(t \to \tau) = (\mathit{Erase}(t) \to \mathit{Erase}(\tau))^1$ (if $\mathit{Proc}(t \to \tau) \neq \emptyset$) $\mathit{Erase}(t) \to \mathit{Erase}(\tau)$ (otherwise); $\mathit{Erase}(\Pi x\colon \sigma.\tau) = (\mathit{Erase}(\sigma) \to$

$Erase(\tau))^1$ (if $Proc(\Pi x\colon \sigma.\tau) \neq \emptyset$) $Erase(\sigma) \to Erase(\tau)$ (otherwise); $Erase(![\Pi(\tilde{x}\colon \tilde{\sigma})t];\alpha) = ![Erase(t)];Erase(\alpha)$, $Erase(![\Pi(\tilde{x}\colon \tilde{\sigma})];\alpha) = ![Erase(\sigma_n)];Erase(\alpha)$, $Erase(\mathsf{end}) = \mathsf{end}$ and others are homomorphic or dual. $Erase(\Gamma)$ is defined homomorphicly except deleting session typings (i.e. $Erase(\Gamma, u\colon \alpha) = Erase(\Gamma)$).

- $Proc(\Delta) = \{k : Erase(\alpha) \mid k : \alpha \in \Delta\}$; $Proc(\mathtt{unit}) = \mathtt{unit}$, $Proc(t \to \tau) = Proc(\tau) \setminus Proc(t)$ and $Proc(\Pi x\colon \sigma.\tau) = Proc(\tau) \setminus x$.
- $Lin(\Gamma) = \{x \mid Erase(\Gamma(x)) = (t)^1\}$.

Next we re-formulate the rules for the arrow types to ensure that all session capabilities are not lost during β-reductions (which is a property of the first system): $t \to \tau$ is well-formed if $Proc(t) \subseteq Proc(\tau)$; and $\Pi x\colon \alpha.\tau$ is so if $x\colon \alpha \in Proc(\tau)$. We also replace $\Delta \cdot k\colon \alpha$ to mean $k \notin \mathsf{fn}(\Delta) \cup \mathsf{fv}(\Delta)$ in the rules for processes. Then we can describe the corresponding side conditions directly using $Proc(\tau)$ and $Lin(\Gamma)$ instead of recording Σ and $\mathcal{S}$. Then we have:

Theorem 4.3 (Embedding). *Below $\Gamma \restriction \mathcal{S}$ means $\{u : \Gamma(u) \mid u \in \mathcal{S}\}$.*

- *Suppose $\Gamma \vdash P : \tau$ is derived by the restricted system defined in this subsection. Then we have: $Erase(\Gamma); Proc(\tau) \setminus \Sigma; \mathcal{S} \vdash Erase(P) : Erase(\tau)$ where $\mathcal{S} = Lin(\Gamma \restriction fv(P))$ and $\Sigma = \{Proc(t) \mid x : t \in \Gamma \restriction \mathcal{S}\}$.*
- *Suppose $\Gamma; \emptyset; \emptyset \vdash P : \diamond$ and P is initial. Then there exist Γ', P' and Δ such that $\Gamma' \vdash P' : \Delta$ with $Erase(\Gamma') = \Gamma$, $Erase(P') = P$ and $\Delta \subset \Gamma'$ in the restricted system.*

The first statement means that the session capabilities of P except those that appear in types of the linear variables in P are placed as Σ, and the linear variables in P are placed as $\mathcal{S}$ in the first system. The proof is by induction on $\Gamma \vdash P : \tau$. The second statement is by constructing the minimum environments starting from $(), x, u$ and $\mathbf{0}$. This theorem shows that the second system (with appropriate use of dependency type information) has a wider typability than the first one, but needs more complex types in the user program. Type inference along the line of [8, 30] may partly ease these burdens, while it is unknown for programs without type annotations.

5 Related and Future Work

This paper studies session types for higher-order processes using two different approaches and compares their typability. The robust formulations hinted by the linear and dependent λ-type theories [3, 28] lead to new process typing systems for protocol validation. Straightforward extensions are recursive types [18, 32], subtyping [14, 30] and polymorphism [12, 26]. In particular, recursive session types are useful to type various common "repetitive" protocols appeared in many practices [7, 10, 29]. For this extension, an explicit recursion construct in the form of the recursive agent $\mathtt{def}\ X(\tilde{x}\tilde{k}) = P\ \mathtt{in}\ M$ is introduced in [18, 32]. In our calculus, this agent can be replaced by a more familiar syntax such as $\mathtt{letrec}\ x = P\ \mathtt{in}\ M$. The important constraint is that P cannot hold linear variables nor free session channels (i.e. $\Sigma = \mathcal{S} = \emptyset$), which does not reduce the expressivity by using parameterised processes as in [28]. By taking the approach in [32], we can construct the typing rule for the recursive agent, and can type scenarios with repetition and recursion which are common in the literature, fully integrated with code mobility, see [1].

There is a large literature on linear and session types for both the λ-calculus and the π-calculus. Below we give the most closely related work, focusing on the linear typing system of the λ-calculus and on the session types for distribution and functional programming languages. See also [1, 6–8, 29] for discussions on other type disciplines of the π-calculus as well as on applications of session types.

Our first typing system is substructural [28] in the sense that for session environments Σ we do not allow weakening, ensuring that a session channel is recorded as having been used only when it actually occurs in session communication expressions; contraction is also not allowed in Σ. Similarly no structural transformations can apply to linear variable environments, ensuring that the occurrence of a variable manifests that it has indeed been used exactly once. The ways in which our typing system enforces linearity can be seen as an amalgamation of the two approaches in [28], retaining the simplicity of declarative systems, and the decidability of algorithmic ones. Contrary to the systems of [28], there is no need to consider linear usage for types other than functional. Applying the techniques in [8, 25], constructing its type inference system would be a straightforward task.

Relating to distribution, [11] studies session types for boxed ambients, preventing session interruption when an ambient crosses its boundary. One of the technical challenges of our work is to formalise sound typing systems for arbitrarily parameterised processes (i.e. λ-abstractions in processes with the full type hierarchy), which is not treated in ambient primitives. In [26, 27] the authors extend previous work [16], and define a concurrent multi-threaded functional language with session primitives. It has explicit multi-threading primitive `fork` and explicit stores whose operational semantics. Their recent draft paper [15] further extends the language to a variant of session types where sending messages is non-blocked. This is handled by explicitly storing an entry for the two endpoint channels in a buffer. It's functionality is the same as our use of two session channels indexed by 0 and 1 for distinguishing two endpoints (based on [14]). They simplify their previous type judgement which requires input and output environments in [26, 27] by using the linear typing with `split` operator, which is more directly related to the original non-deterministic typing [28]. While a precise typability comparison is difficult due to our additional language primitives and their operational semantics with buffers (which is essential for type soundness in their language), their work also shows a use of the linear types for functional languages with sessions. Our comparison between the first and second systems via Theorem 4.3 makes the relationship between controlling usage of functional variables and effects of channel accessibility clear: the idea of "balanced" seems more suited to effect-like systems since our concern is well-formedness of process types, not intermediate functional types, while the linear typing approach is simpler and more tractable. This line of study is not explored in the previous literature.

As an on-going work, we have been investigating the incorporation of session types and code mobility with Sockets in Java [19] and Web Service Description Languages [7, 29]. From these experiences, we find that not only type checking by session types after writing a protocol, but also declaring its session types before compilation, greatly helps programmers implement error-free interactions in their programs. For developing programming language designs, the presented type theory needs further explorations, including its incorporation with advanced concurrent programming primitives such as exceptions, timeout and priority checking.

References

1. On-line Appendix of this paper. http://www.doc.ic.ac.uk/~yoshida/hopis.
2. A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. In *OOPSLA '05*, pages 403–422. ACM Press, 2005. A full version will appear in *TCS*.
3. D. Aspinall and M. Hofmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT, 2005. Editor Benjamin C. Pierce.
4. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. of Funct. Progr.*, 15(2):219–248, 2005.
5. M. Carbone, K. Honda, and N. Yoshida. A Calculus of Global Interaction Based on Session Types. In *DMC'06*, ENTCS. Elsevier, 2006. to appear.

6. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, LNCS. Springer, 2007. to appear.
7. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. To be published by W3C. Available at http://www.dcs.qmul.ac.uk/∼carbonem/cdlpaper/workingnote.pdf, 2006.
8. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
9. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object Oriented Language with Session Types. In *TGC*, volume 3705 of *LNCS*, pages 299–318, 2005.
10. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
11. P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In M. Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.
12. S. Gay. Bounded polymorphism in session types. *MSCS*. To appear.
13. S. Gay and M. Hole. Types and Subtypes for Client-Server Interactions. In *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.
14. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Info.*, 42(2/3):191–225, 2005.
15. S. Gay and V. T. Vasconcelos. A new approach to functional session types, October 2006. A Draft.
16. S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.
17. M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: A language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.
18. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.
19. R. Hu. Implementation of a distributed mobile Java. Master's thesis, Imperial College London, 2006.
20. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.
21. M. Neubauer and P. Thiemann. An implementation of session types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
22. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.
23. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
24. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Objects and Components using Session Types. In *FOCLASA'02*, volume 68(3) of *ENTCS*. Elsevier, 2002.
25. V. T. Vasconcelos. A note on a typing system for the higher-order π-calculus. Keio University, Sept. 1993.
26. V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *Theor. Comp. Sci.*, 2006. To appear.
27. V. T. Vasconcelos, A. Ravara, and S. Gay. Session Types for Functional Multithreading. In *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
28. D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT, 2005. Editor Benjamin C. Pierce.
29. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.
30. N. Yoshida. Channel dependency types for higher-order mobile processes. In *POPL '04*, pages 147–160. ACM Press, 2004. Full version available at www.doc.ic.ac.uk/˜yoshida.
31. N. Yoshida and M. Hennessy. Assigning types to processes. *Info. & Comp.*, 172:82–120, 2002.
32. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisit. In *SecRet'06*, ENTCS. Elsevier. To appear.

A Appendix: Section 2

We define the structural congruence omitted from the main section. The relation, denoted '$\equiv$', is the smallest relation generated by the following axioms and rules.

$P =_\alpha Q \Rightarrow P \equiv Q$	Renaming of bound variables
$P \mid Q \equiv Q \mid P$	Commutativity of parallel composition
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	Associativity of parallel composition
$P \mid \mathbf{0} \equiv P$	Inaction and parallel composition
$(\nu a:\sigma)P \mid Q \equiv (\nu a:\sigma)(P \mid Q) \quad a \notin \mathsf{fn}(Q)$	Scope extrusion
$(\nu\kappa)P \mid Q \equiv (\nu\kappa)(P \mid Q) \quad \kappa_{i\in\{0,1\}} \notin \mathsf{fn}(Q)$	
$(\nu a:\sigma)(\nu\kappa)P \equiv (\nu\kappa)(\nu a:\sigma)P$	Exchange
$(\nu a:\sigma)(\nu b:\sigma')P \equiv (\nu b:\sigma')(\nu a:\sigma)P$	
$(\nu\kappa)(\nu\kappa')P \equiv (\nu\kappa')(\nu\kappa)P$	
$(\nu a:\sigma)\mathbf{0} \equiv \mathbf{0} \qquad (\nu\kappa)\mathbf{0} \equiv \mathbf{0}$	Inaction and restriction

B Appendix: Section 4

This appendix lists the omitted definitions and rules in Section 4. First we define the set of free names and variables as follows.

(Free Variables)

$\mathsf{fv}(x) = \{x\},\ \mathsf{fv}(a) = \mathsf{fv}(\kappa_i) = \emptyset.$

$\mathsf{fv}(\mathsf{unit}) = \mathsf{fv}(\mathsf{end}) = \emptyset$

$\mathsf{fv}(t \to \tau) = \mathsf{fv}(t) \cup \mathsf{fv}(\tau)$

$\mathsf{fv}(\Pi x{:}\,\sigma.\tau) = \mathsf{fv}(\sigma) \cup \mathsf{fv}(\tau) \setminus x$

$\mathsf{fv}(\mathsf{begin}.\alpha) = \mathsf{fv}(\sigma)$

$\mathsf{fv}(\Delta) = \{\mathsf{fv}(u) \cup \mathsf{fv}(\tau) \mid u{:}\,\tau \in \Delta\}$

$\mathsf{fv}(!\,[\Pi(x_1{:}\,\sigma_1,\ldots,x_n{:}\,\sigma_n)\tau_1,\ldots,\tau_m]\alpha)$
$\quad = \mathsf{fv}(?\,[\Pi(x_1{:}\,\sigma_1,\ldots,x_n{:}\,\sigma_n)\tau_1,\ldots,\tau_m]\alpha)$
$\quad = (\mathsf{fv}(\sigma_1)\ldots \cup \mathsf{fv}(\sigma_n) \cup \mathsf{fv}(\tau_1)\ldots \cup \mathsf{fv}(\tau_m)) \setminus \tilde{x} \cup \mathsf{fv}(\alpha)$

$\mathsf{fv}(\&[l_1{:}\,\alpha_1;\ldots l_n{:}\,\alpha_n]) = \mathsf{fv}(\oplus[l_1{:}\,\alpha_1;\ldots l_n{:}\,\alpha_n]) = \cup\mathsf{fv}(\alpha_i)$

(Free Names)

$\mathsf{fn}(x) = \emptyset,\ \mathsf{fn}(a) = \{a\},\ \mathsf{fn}(\kappa_i) = \{\kappa_i\}$

$\mathsf{fn}(\Pi x{:}\,\sigma.\tau) = \mathsf{fn}(\sigma) \cup \mathsf{fn}(\tau)$

$\mathsf{fn}(!\,[\Pi(x_1{:}\,\sigma_1,\ldots,x_n{:}\,\sigma_n)\tau_1,\ldots,\tau_m]\alpha)$
$\quad = \mathsf{fn}(?\,[\Pi(x_1{:}\,\sigma_1,\ldots,x_n{:}\,\sigma_n)\tau_1,\ldots,\tau_m]\alpha)$
$\quad = (\mathsf{fn}(\sigma_1)\ldots \cup \mathsf{fn}(\sigma_n) \cup \mathsf{fn}(\tau_1)\ldots \cup \mathsf{fn}(\tau_m)) \cup \mathsf{fv}(\alpha)$

Other rules are given by replacing $\mathsf{fv}(\,)$ by $\mathsf{fn}(\,)$.

Well-Formedness Next we define the notions of well-formedness for types and environments by the formal system in Figure 6. As in [30, 31], the first judgement $\Gamma \vdash \mathtt{Env}$ is designed to ensure that an identifier can only be used in the construction of a type if it has already been *declared* in the environment. (e-base) and (e-chan) are standard. The formation rule for functional types are the same as [30, 31]; (t-base) is for constant and end type, and (t-abs$_H$) is for higher-order values. In the formation rule for dependent types, (t-abs$_N$), the bound variable x is allowed in the construction of the result type τ. (t-dep) for channel dependent types is similarly defined. (t-sel) is for selection. Then (t-dual) formulates the input types. For process types, (t-proc) ensures that a process always has a channel environment which does not exceed the capability specified by the global environment Γ, see [31, § 3.1].

(Well-Formed Environment)

$$\text{(e-nil)}\quad \emptyset \vdash \texttt{Env} \qquad \text{(e-val)}\ \frac{\Gamma \vdash t : \texttt{tp} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x\colon t \vdash \texttt{Env}} \qquad \text{(e-chan)}\ \frac{\Gamma \vdash \sigma : \texttt{tp} \quad u \notin \mathrm{dom}(\Gamma)}{\Gamma, u\colon \sigma \vdash \texttt{Env}}$$

(Well-Formed Types)

$$\text{(t-base)}\ \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \texttt{unit}, \texttt{end} : \texttt{tp}} \qquad \text{(t-abs}_H\text{)}\ \frac{\Gamma \vdash t\colon \texttt{tp} \quad \Gamma \vdash \tau\colon \texttt{tp}}{\Gamma \vdash t \to \tau : \texttt{tp}} \qquad \text{(t-abs}_N\text{)}\ \frac{\Gamma, x\colon \sigma \vdash \tau\colon \texttt{tp}}{\Gamma \vdash \Pi x\colon \sigma.\tau : \texttt{tp}}$$

$$\text{(t-proc)}\ \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \emptyset : \texttt{tp}} \qquad \frac{\begin{array}{c}\Gamma \vdash \Delta : \texttt{tp} \quad \Gamma \vdash u : \sigma \\ u \notin \mathrm{dom}(\Delta)\end{array}}{\Gamma \vdash \Delta, u : \sigma : \texttt{tp}} \qquad \text{(t-beg)}\ \frac{\Gamma \vdash \alpha : \texttt{tp}}{\Gamma \vdash \mathsf{begin}.\alpha : \texttt{tp}} \qquad \text{(t-dual)}\ \frac{\Gamma \vdash \sigma\colon \texttt{tp}}{\Gamma \vdash \overline{\sigma}\colon \texttt{tp}}$$

$$\text{(t-out)}\ \frac{\Gamma, x_1\colon \sigma_1, .., x_n\colon \sigma_n \vdash \tau_j\colon \texttt{tp} \quad \Gamma \vdash \alpha : \texttt{tp}}{\Gamma \vdash !\lceil \Pi(x_1\colon \sigma_1, ..., x_n\colon \sigma_n)\tau_1, ..., \tau_m\rceil; \alpha : \texttt{tp}} \qquad \text{(t-sel)}\ \frac{\Gamma \vdash \tau_j\colon \texttt{tp}}{\Gamma \vdash \oplus\{l_1\colon \alpha_1, ..., l_n\colon \alpha_n\} : \texttt{tp}}$$

Fig. 6. Well-Formed Higher-Order IO Types