Type-Based Analysis for Session Inference (Extended Abstract)

Carlo Spaccasassi, Vasileios Koutavas

Type-Based Analysis for Session Inference (Extended Abstract)*

Carlo Spaccasassi and Vasileios Koutavas

Trinity College Dublin

Abstract. We propose a type-based analysis to infer the session protocols of channels in an ML-like concurrent functional language. Combining and extending well-known techniques, we develop a type-checking system that separates the underlying ML type system from the typing of sessions. Without using linearity, our system guarantees communication safety and partial lock freedom. It also supports provably complete session inference for finite sessions with no programmer annotations. We exhibit the usefulness of our system with interesting examples, including one which is not typable in substructural type systems.

1 Introduction

Concurrent programming often requires processes to communicate according to intricate protocols. In mainstream programming languages these protocols are encoded implicitly in the program's control flow, and no support is available for verifying their correctness.

Honda [6] first suggested the use of *binary session types* to explicitly describe and check protocols over communication channels with two endpoints. Fundamentally, session type systems guarantee that a program respects the order of communication events (session fidelity) and message types (communication safety) described in a channel's session type. A number of session type systems (e.g., [2, 3, 16]) also ensure that processes fully execute the protocols of their open endpoints, as long as they do not diverge or block on opening new sessions (partial lock freedom).

To date, binary session type disciplines have been developed for various process calculi and high-level programming languages (see [8] for an overview) by following one of two main programming language design approaches: using a single substructural type system for both session and traditional typing [5, 7, 18, 19], or using monads to separate the two [13, 16].

In this paper we propose a third design approach which uses *effects*. Similar to previous work, our approach enables the embedding of session types in programming languages with sophisticated type systems. Here we develop a high-level language where intricate protocols of communication can be programmed and checked statically (Sect. 2). Contrary to both monads and substructural type

* This research was supported, in part, by Science Foundation Ireland grant 13/RC/2094. The first author was supported by MSR (MRL 2011-039).

systems, our approach allows pure code to call library code with communication effects, without having to refactor the pure code (e.g., to embed it in a monad or pass continuation channels through it—see Ex. 2.3). We apply our approach to $\mathrm{ML_S}$, a core of ML with session communication (Sect. 3).

Our approach separates traditional typing from session typing in a two-level system, which follows the principles of *typed based analysis* [12]. The first level employs a type-and-effect system, which adapts and extends the one of Amtoft, Nielson and Nielson [1] to session communication (Sect. 4). At this level the program is typed against an ML type and a *behaviour* which abstractly describes program structure and communication. Session protocols are not considered here—they are entirely checked at the second level. Thus, each endpoint is given type Ses^{ρ}, where ρ statically approximates its source. The benefit of extending [1] is that we obtain a complete behaviour inference algorithm, which extracts a behaviour for every program respecting ML types.

At the second level, our system checks that a behaviour, given an operational semantics, complies with the session types of channels and endpoints (Sect. 5). The session discipline realised here is inspired by the work of Castagna et al. [3]. This discipline guarantees that programs comply with session fidelity and communication safety, but also, due to stacked interleaving of sessions, partial lock freedom. However, one of the main appeals of our session typing discipline is that it enables a provably *complete session types inference* from behaviours which, with behaviour inference, gives us a complete method for session inference from $\mathrm{ML_S}$, without programmer annotations (Sect. 6). The two levels of our system only interact through behaviours, which we envisage will allow us to develop front-ends for different languages and back-ends for different session disciplines.

To simplify the technical development we consider only sessions of finite interactions. However, we allow recursion in the source language, as long as it is *confined*: recursive code may only open new sessions and completely consume them (see Sect. 2). In Sect. 7 we discuss an extension to recursive types. Related work and conclusions can be found in Sect. 8. Details missing from this extended abstract can be found in the appendix for the benefit of the reviewers.

2 Motivating Examples

Example 2.1 (A Swap Service). A coordinator process uses the primitive acc-swp to accept two connections on a channel swp (we assume functions acc-c and req-c for every channel c), opening two concurrent sessions with processes that want to exchange values. It then coordinates the exchange and recurs.

```
let fun coord(_) =
  let val p1 = acc-swp ()
      val x1 = recv p1
      val p2 = acc-swp ()
      val x2 = recv p2
  in send p2 x1; send p1 x2; coord ()
in spawn coord;
```

```
let fun swap(x) =
  let val p = req-swp ()
  in send p x; recv p
in spawn (fn _ => swap 1);
   spawn (fn _ => swap 2);
```

Each endpoint the coordinator receives from calling acc-swp are used according to the session type $?T.!T.\mathsf{end}$. This says that, on each endpoint, the coordinator will first read a value type T ($?T$), then output a value of the same type ($!T$) and close the endpoint (end). The interleaving of sends and receives on the two endpoints achieves the desired swap effect.

Function swap : Int$\to T'$ calls req-swp and receives and endpoint which is used according to the session type $!\mathsf{Int}.?T'.\mathsf{end}$. By comparing the two session types above we can see that the coordinator and the swap service can communicate without type errors, and indeed are typable, when $T = \mathsf{Int} = T'$. Our type inference algorithm automatically deduces the two session types from this code.

Because swp is a global channel, ill-behaved client code can connect to it too:

```
let val p1 = req-swp () in send p1 1;
let val p2 = req-swp () in send p2 2;
let val (x1, x2) = (recv p1, recv p2) in e_cl
```

This client causes a deadlock, because the coordinator first sends on p2 and then on p1, but this code orders the corresponding receives in reverse. The interleaving of sessions in this client is rejected by our type system because it is not *well-stacked*: recv p1 is performed before the most recent endpoint (p2) is closed. The interleaving in the coordinator, on the other hand, is well-stacked.

Example 2.2 (Delegation for Efficiency).

In the previous example the coordinator is a bottleneck when exchanged values are large. A more efficient implementation delegates exchange to the clients:

```
let fun coord(_) =
  let val p1 = acc-swp ()
  in sel-SWAP p1;
     let val p2 = acc-swp
     in sel-LEAD p2;
        deleg p2 p1;
        coord()
```

```
let fun swap(x) =
  let val p = req-swp ()
  in case p {
     SWAP: send p x; recv p
     LEAD: let val q = resume p
               val y = recv q
           in send q x; y }
```

Function swap again connects to the coordinator over channel swp, but now offers two choices with the labels SWAP and LEAD. If the coordinator selects the former, the swap method proceeds as before; if it selects the latter, swap resumes (i.e., inputs) another endpoint, binds it to q, and performs a rcv and then a send on q. The new coordinator accepts two sessions on swp, receiving two endpoints: p1 and p2. It selects SWAP on p1, LEAD on p2, sends p1 over p2 and recurs.

When our system analyses the coordinator in isolation, it infers the protocol $\eta_{\mathsf{coord}} = (!\mathsf{SWAP}.\eta' \oplus !\mathsf{LEAD}.!\eta'.\mathsf{end})$ for both endpoints p1 and p2. When it analyses swap : $T_1 \to T_2$, it infers $\eta_{\mathsf{p}} = \Sigma\{?\mathsf{SWAP}.!T_1.?T_2.\mathsf{end},\ ?\mathsf{LEAD}.?\eta_{\mathsf{q}}.\mathsf{end}\}$ and $\eta_{\mathsf{q}} = ?T_2.!T_1.\mathsf{end}$ as the protocols of p and q, respectively. The former *selects* either options SWAP or LEAD and the latter *offers* both options.

If the coordinator is type-checked in isolation, then typing succeeds with any η': the coordinator can delegate any session. However, because of duality, the typing of req-swp in the swap function implies that $\eta' = \eta_{\mathsf{q}}$ and $T_1 = T_2$. Our inference algorithm can type this program and derive the above session types.

Example 2.3 (A Database Library). In this example we consider the implementation of a library which allows clients to connect to a database.

```
let fun coord(_) =
  let val p = acc-db ()
      fun loop(_) = case p {
        QRY: let val sql = recv p
                 val res = process sql
             in send p res; loop ()
        END: () }
  in spawn coord; loop ()
in spawn coord;
```

```
let fun clientinit (_) =
  let val con = req-db ()
      fun query(sql) = sel-QRY con;
                       send con sql;
                       recv con
      fun close(_) = sel-END con
  in (query, close)
in e_client
```

The coordinator accepts connections from clients on channel db. If a connection is established, after spawning a copy of the coordinator to serve other clients, the coordinator enters a loop that serves the connected client. In this loop it offers the client two options: QRY and END. If the client selects QRY, the coordinator receives an SQL query, processes it (calling process : sql → dbresult, with these types are defined in the library), sends back the result, and loops. If the client selects END the connection with the coordinator closes and the current coordinator process terminates.

Function clientinit is exposed to the client, which can use it to request a connection with the database coordinator. When called, it establishes a connection con and returns two functions to the client: query and close. Then, the client code e_{client} can apply the query function to an sql object and receive a dbresult as many times as necessary, and then invoke close to close the connection. Using our two-level inference system with recursion Sect. 7, we can infer the session type of the coordinator's endpoint p: $\mu X.\Sigma\{?\mathsf{QRY}.?\mathsf{sql}.!\mathsf{dbresult}.X,\ ?\mathsf{END}.\mathsf{end}\}$, and check whether the client code e_{client} respects it.

This example is not typable with a substructural type system because query and close share the same (linear) endpoint con. Moreover, in a monadic system e_{client} will need to be converted to monadic form.

3 Syntax and Operational Semantics of $\mathrm{ML_S}$

Fig. 1 shows the syntax and operational semantics of $\mathrm{ML_S}$, a core of ML with session communication. An expression can be one of the usual lambda expressions or spawn e which evaluates e to a function and asynchronously applies it to the unit value; it can also be case $e\,\{L_i : e_i\}_{i\in I}$ which, as we will see, implements finite *external choice*. We use standard syntactic sugar for writing programs. A *system* S is a parallel composition of closed expressions *(processes)*.

The operational semantics of $\mathrm{ML_S}$ are standard; here we only discuss session-related rules. Following the tradition of binary session types [7], communication between processes happens over dynamically generated entities called *sessions* which have exactly two *endpoints*. Thus, $\mathrm{ML_S}$ values contain a countably infinite set of endpoints, ranged over by p. We assume a total involution $(\overline{\cdot})$ over this set, with the property $\overline{p} \neq p$, which identifies *dual endpoints*.

$$
\begin{array}{lll}
\textbf{Exp:} & e ::= v \mid (e,e) \mid e\,e \mid \mathsf{let}\,x = e\,\mathsf{in}\,e \mid \mathsf{if}\,e\,\mathsf{then}\,e\,\mathsf{else}\,e \mid \mathsf{spawn}\,e \mid \mathsf{case}\,e\,\{L_i : e_i\}_{i\in I} \\
\textbf{Sys:} & S ::= e \mid S \parallel S \\
\textbf{Val:} & v ::= x \mid k \in \mathsf{Const} \mid (v,v) \mid \mathsf{fn}\,x \Rightarrow e \mid \mathsf{fun}\,f(x) = e \mid p \\
 & \quad \mid \mathsf{req}\text{-}c \mid \mathsf{acc}\text{-}c \mid \mathsf{send} \mid \mathsf{recv} \mid \mathsf{sel}\text{-}L \mid \mathsf{deleg} \mid \mathsf{resume} \\
\textbf{ECxt:} & E ::= [\cdot] \mid (E,e) \mid (v,E) \mid E\,e \mid v\,E \mid \mathsf{let}\,x = E\,\mathsf{in}\,e \mid \mathsf{if}\,E\,\mathsf{then}\,e\,\mathsf{else}\,e \\
 & \quad \mid \mathsf{spawn}\,E \mid \mathsf{case}\,E\,\{L_i : e_i\}_{i\in I}
\end{array}
$$

$$
\begin{array}{lll}
\textsc{RIft} & \mathsf{if\,tt\,then}\,e_1\,\mathsf{else}\,e_2 \hookrightarrow e_1 & \textsc{RLet} \quad \mathsf{let}\,x = v\,\mathsf{in}\,e \hookrightarrow e[v/x] \\
\textsc{RIff} & \mathsf{if\,ff\,then}\,e_1\,\mathsf{else}\,e_2 \hookrightarrow e_2 & \textsc{RFix} \quad (\mathsf{fun}\,f(x) = e)\,v \hookrightarrow e[\mathsf{fun}\,f(x) = e/f][v/x]
\end{array}
$$

$$
\begin{array}{lll}
\textsc{RBeta} & E[e] \parallel S \longrightarrow E[e'] \parallel S & \text{if } e \hookrightarrow e' \\
\textsc{RSpn} & E[\mathsf{spawn}\,v] \parallel S \longrightarrow E[()] \parallel v\,() \parallel S & \\
\textsc{RInit} & E_1[\mathsf{req}\text{-}c\,()] \parallel E_2[\mathsf{acc}\text{-}c\,()] \parallel S \longrightarrow E_1[p] \parallel E_2[\overline{p}] \parallel S & \text{if } p, \overline{p} \text{ fresh} \\
\textsc{RCom} & E_1[\mathsf{send}\,(p,v)] \parallel E_2[\mathsf{recv}\,\overline{p}] \parallel S \longrightarrow E_1[()] \parallel E_2[v] \parallel S & \\
\textsc{RDel} & E_1[\mathsf{deleg}\,(p,p')] \parallel E_2[\mathsf{resume}\,\overline{p}] \parallel S \longrightarrow E_1[()] \parallel E_2[p'] \parallel S & \\
\textsc{RSel} & E_1[\mathsf{sel}\text{-}L_j\,p] \parallel E_2[\mathsf{case}\,\overline{p}\,\{L_i : e_i\}_{i\in I}] \parallel S \longrightarrow E_1[()] \parallel E_2[e_j] \parallel S & \text{if } j \in I
\end{array}
$$

Fig. 1. $\mathrm{ML_S}$ syntax and operational semantics.

A process can request (or accept) a new session by calling $\mathsf{req}\text{-}c$ (resp., $\mathsf{acc}\text{-}c$) with the unit value, which returns the endpoint (resp., dual endpoint) of a new session. Here c ranges over an infinite set of global initialisation *channels*. To simplify presentation, the language contains $\mathsf{req}\text{-}c$ and $\mathsf{acc}\text{-}c$ for each channel c.

Once two processes synchronise on a global channel and each receives a fresh, dual endpoint (RInit reduction), they can exchange messages (RCom), *delegate* endpoints (RDel) and offer a number of choices $L_{i\in I}$, from which the partner can select one (RSel). Here L ranges over a countably infinite set of choice labels, and I is a finite set of natural numbers; L_i denotes a unique label for each natural number i and we assume $\mathsf{sel}\text{-}L_i$ for each L_i.

The next two sections present the two-level type system of $\mathrm{ML_S}$.

4 First Level Typing: ML Typing and Behaviours

Here we adapt and extend the type-and-effect system of Amtoft, Nielson and Nielson [1] to session communication in $\mathrm{ML_S}$. A judgement $C;\ \Gamma \vdash e : T \triangleright b$ states that e has type T and behaviour b, under type environment Γ and *constraint environment* C. The constraint environment relates type-level variables to terms and enables type inference. These components are defined in Fig. 2.

An $\mathrm{ML_S}$ expression can have a standard type or an endpoint type Ses^{ρ}. Function types are annotated with a behaviour variable β. *Type variables* α are used for ML polymorphism. As in [1], Hindley-Milner polymorphism is extended with type schemas TS of the form $\forall(\vec{\gamma}{:}C_0).\,T$, where γ ranges over variables $\alpha, \beta, \rho, \psi$, and C_0 imposes constraints on the quantified variables with $\mathrm{fv}(C_0) \subseteq \{\vec{\gamma}\}$. Type environments Γ bind program variables to type schemas; we let $\forall(\emptyset).T = T$.

The rules of our type-and-effect system are shown in Fig. 3 which, as in [1, Sec. 2.8], is a conservative extension of ML. This system performs both ML

$$
\begin{array}{rl}
\textbf{Variables:} & \alpha\textbf{(Type)} \quad \beta\textbf{(Behaviour)} \quad \psi\textbf{(Session)} \quad \rho\textbf{(Region)} \\
\textbf{T. Schemas:} & TS ::= \forall(\vec{\alpha}\vec{\beta}\vec{\rho}\vec{\psi} : C).\, T \qquad\qquad \textbf{Regions:}\ \ r ::= l \mid \rho \\
\textbf{Types:} & T ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid T \times T \mid T \xrightarrow{\beta} T \mid \mathsf{Ses}^{\rho} \mid \alpha \\
\textbf{Constraints:} & C ::= T \subseteq T \mid \mathit{cfd}(T) \mid b \subseteq \beta \mid \rho \sim r \mid c \sim \eta \mid \overline{c} \sim \eta \mid \eta \bowtie \eta \mid C, C \mid \epsilon \\
\textbf{Behaviours:} & b ::= \beta \mid \tau \mid b\,;b \mid b \oplus b \mid \mathsf{rec}_{\beta}\, b \mid \mathsf{spawn}\, b \mid \mathsf{push}(l : \eta) \\
 & \quad \mid \rho!T \mid \rho?T \mid \rho!\rho \mid \rho?l \mid \rho!L_i \mid \underset{i \in I}{\&}\{\rho?L_i\,;b_i\} \\
\textbf{Type Envs:} & \Gamma ::= x : TS \mid \Gamma, \Gamma \mid \epsilon
\end{array}
$$

Fig. 2. Syntax of types, behaviours, constraints, and session types.

type checking (including type-schema inference), and behaviour checking (which enables behaviour inference). Rules TLet, TVar, TIf, TConst, TApp, TFun, TSpawn and the omitted rule for pairs perform standard type checking and straightforward sequential ($b_1\,;b_2$) and non-deterministic ($b_1 \oplus b_2$) behaviour composition; τ is the behaviour with no effect.

Just as a type constraint $T \subseteq \alpha$ associates type T with type variable α, a behaviour constraint $b \subseteq \beta$ associated behaviour b to behaviour variable β. Intuitively, β is the non-deterministic composition of all its associated behaviours. Rule TSub allows the replacement of behaviour b with variable β; such replacement in type annotations yields a subtyping relation ($C \vdash T <: T'$). Rules TIns and TGen are taken from [1] and extend ML's type schema instantiation and generalisation rules, respectively. Because we extend Hindley-Milner's let polymorphism, generalisation (TGen) is only applied to the right-hand side expression of the let construct. The following definition allows the instantiation of a type schema under a global constraint environment C. We write $C \vdash C'$ when C' is included in the reflexive, transitive, compatible closure of C.

Definition 4.1 (Solvability). $\forall(\vec{\gamma} : C_0).\, T$ *is* solvable *by* C *and substitution* σ *when* $\mathit{dom}(\sigma) \subseteq \{\vec{\gamma}\}$ *and* $C \vdash C_0\sigma$.

In TRec, the communication effect of the body of a recursive function should be *confined*, which means it may only use endpoints it opens internally. For this reason, the function does not input nor return open endpoints or other non-confined functions ($C \vdash \mathit{confd}(T, T')$). Although typed under Γ which may contain endpoints and non-confined functions, the effect of the function body is recorded in its behaviour. The second level of our system checks that if the function is called, no endpoints from its environment are affected. It also checks that the function fully consumes internal endpoints before it returns or recurs.

A type T is confined when it does not contain any occurrences of the endpoint type Ses^{ρ} for any ρ, and when any b in T is confined. A behaviour b is confined when all of its possible behaviours are either τ or recursive.

To understand rule TEndp, we have to explain region variables (ρ), which are related to region constants through C. Region constants are simple program annotations l (produced during pre-processing) which uniquely identify the textual

$$\textsc{TLet}\ \frac{C;\ \Gamma \vdash e_1 : TS \triangleright b_1 \quad C;\ \Gamma, x : TS \vdash e_2 : T \triangleright b_2}{C;\ \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : T \triangleright b_1 ; b_2}$$

$$\textsc{TVar}\ \ C;\ \Gamma \vdash x : \Gamma(x) \triangleright \tau$$

$$\textsc{TIf}\ \frac{C;\ \Gamma \vdash e_1 : \mathsf{Bool} \triangleright b_1 \quad C;\ \Gamma \vdash e_i : T \triangleright b_i\ \ (i \in \{1,2\})}{C;\ \Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : T \triangleright b_1 ; (b_2 \oplus b_3)}$$

$$\textsc{TConst}\ \ C;\ \Gamma \vdash k : \mathit{typeof}(k) \triangleright \tau$$

$$\textsc{TApp}\ \frac{C;\ \Gamma \vdash e_1 : T' \xrightarrow{\beta} T \triangleright b_1 \quad C;\ \Gamma \vdash e_2 : T' \triangleright b_2}{C;\ \Gamma \vdash e_1\, e_2 : T \triangleright b_1 ; b_2 ; \beta}$$

$$\textsc{TFun}\ \frac{C;\ \Gamma, x : T \vdash e : T' \triangleright \beta}{C;\ \Gamma \vdash \mathsf{fn}\, x \Rightarrow e : T \xrightarrow{\beta} T' \triangleright \tau}$$

$$\textsc{TMatch}\ \frac{C;\ \Gamma \vdash e : \mathsf{Ses}^{\rho} \triangleright b \quad C;\ \Gamma \vdash e_i : T \triangleright b_i\ \ (i \in I)}{C;\ \Gamma \vdash \mathsf{case}\, e\, \{L_i : e_i\}_{i \in I} : T \triangleright b ; \underset{i \in I}{\&} \{\rho ? L_i ; b_i\}}$$

$$\textsc{TEndp}\ \ C;\ \Gamma \vdash p^l : \mathsf{Ses}^{\rho} \triangleright \tau \quad C \vdash \rho \sim l$$

$$\textsc{TSub}\ \frac{C;\ \Gamma \vdash e : T \triangleright b}{C;\ \Gamma \vdash e : T' \triangleright \beta} \quad \begin{array}{l} C \vdash T <: T' \\ C \vdash b \subseteq \beta \end{array}$$

$$\textsc{TSpawn}\ \frac{C;\ \Gamma \vdash e : \mathsf{Unit} \xrightarrow{\beta} \mathsf{Unit} \triangleright b}{C;\ \Gamma \vdash \mathsf{spawn}\, e : \mathsf{Unit} \triangleright b ; \mathsf{spawn}\, \beta}$$

$$\textsc{TRec}\ \frac{C;\ \Gamma, f : T \xrightarrow{\beta} T', x : T \vdash e : T' \triangleright b}{C;\ \Gamma \vdash \mathsf{fun}\, f(x) = e : T \xrightarrow{\beta} T' \triangleright \tau} \quad \begin{array}{l} C \vdash \mathit{confd}(T, T') \\ C \vdash \mathsf{rec}_{\beta}\, b \subseteq \beta \end{array}$$

$$\textsc{TIns}\ \frac{C;\ \Gamma \vdash e : \forall(\vec{\gamma} : C_0).T \triangleright b}{C;\ \Gamma \vdash e : T\sigma \triangleright b} \quad \begin{array}{l} \mathit{dom}(\sigma) \subseteq \{\vec{\gamma}\} \\ \forall(\vec{\gamma} : C_0).\, T \text{ is solvable by } C \text{ and } \sigma \end{array}$$

$$\textsc{TGen}\ \frac{C \cup C_0;\ \Gamma \vdash e : T \triangleright b}{C;\ \Gamma \vdash e : \forall(\vec{\gamma} : C_0).T \triangleright b} \quad \begin{array}{l} \{\vec{\gamma}\} \cap \mathrm{fv}(\Gamma, C, b) = \emptyset \\ \forall(\vec{\gamma} : C_0).T \text{ is solvable by } C \text{ and some } \sigma \end{array}$$

Fig. 3. Type-and-Effect System for $\mathrm{ML_S}$ Expressions (omitting rule for pairs).

sources of endpoints. We thus type an extended $\mathrm{ML_S}$ syntax

$$\textbf{Values:} \qquad v ::= \ldots \mid p^l \mid \mathsf{req}\text{-}c^l \mid \mathsf{acc}\text{-}c^l \mid \mathsf{resume}^l$$

If a sub-expression has type Ses^{ρ} and it evaluates to a value p^l, then it must be that $C \vdash \rho \sim l$, denoting that p was generated from the code location identified by l. This location will contain one of $\mathsf{req}\text{-}c^l$, $\mathsf{acc}\text{-}c^l$, or resume^l. These primitive functions (typed by TConst) are given the following type schemas.

$$\begin{array}{ll} \mathsf{req}\text{-}c^l & : \forall(\beta\rho\psi : \mathsf{push}(l : \psi) \subseteq \beta, \rho \sim l, c \sim \psi).\, \mathsf{Unit} \xrightarrow{\beta} \mathsf{Ses}^{\rho} \\ \mathsf{acc}\text{-}c^l & : \forall(\beta\rho\psi : \mathsf{push}(l : \psi) \subseteq \beta, \rho \sim l, \bar{c} \sim \psi).\, \mathsf{Unit} \xrightarrow{\beta} \mathsf{Ses}^{\rho} \\ \mathsf{resume}^l & : \forall(\beta\rho\rho' : \rho ? l \subseteq \beta,\ \ \rho' \sim l).\, \mathsf{Ses}^{\rho} \xrightarrow{\beta} \mathsf{Ses}^{\rho'} \end{array}$$

An application of $\mathsf{req}\text{-}c^l$ starts a new session on the static endpoint l. To type it, C must contain its effect $\mathsf{push}(l : \psi) \subseteq \beta$, where ψ is a session variable, representing the session type of l. At this level session types are ignored (hence the use of a simple ψ); they become important in the second level of our typing system. Moreover, C must record that session variable ρ is related to l ($\rho \sim l$) and that the "request" endpoint of channel c has session type ψ ($c \sim \psi$). The

only difference in the type schema of $\mathsf{acc}\text{-}c^l$ is that the "accept" endpoint of c is related to ψ ($\overline{c} \sim \psi$). Resume receives an endpoint (ρ') over another one (ρ), recorded in its type schema ($\rho?\rho' \subseteq \beta$); ρ is an existing endpoint but ρ' is treated as an endpoint generated by resume^l, hence the constraint $\rho' \sim l$.

The following are the type schemas of the rest of the constant functions.

$$\begin{aligned}
\mathsf{recv} &: \forall(\alpha\beta\rho : \rho?\alpha \subseteq \beta, \mathit{cfd}(\alpha)).\, \mathsf{Ses}^\rho \xrightarrow{\beta} \alpha \\
\mathsf{send} &: \forall(\alpha\beta\rho : \rho!\alpha \subseteq \beta, \mathit{cfd}(\alpha)).\, \mathsf{Ses}^\rho \times \alpha \xrightarrow{\beta} \mathsf{Unit} \\
\mathsf{deleg} &: \forall(\beta\rho\rho' : \rho!\rho' \subseteq \beta).\, \mathsf{Ses}^\rho \times \mathsf{Ses}^{\rho'} \xrightarrow{\beta} \mathsf{Unit} \\
\mathsf{sel}\text{-}L &: \forall(\beta\rho : \rho?L \subseteq \beta).\, \mathsf{Ses}^\rho \xrightarrow{\beta} \mathsf{Unit}
\end{aligned}$$

These record input ($\rho?\alpha$), output ($\rho!\alpha$), delegation ($\rho!\rho'$), or selection ($\rho!L_i$) behaviour. For input and output the constraint $\mathit{cfd}(\alpha)$ must be in C, recording that the α can be instantiated only with confined types.

5 Second Level Typing: Session Types

Session types describe the communication protocols of endpoints; their syntax is:

$$\eta ::= \mathsf{end} \mid !T.\eta \mid ?T.\eta \mid !\eta.\eta \mid ?\eta.\eta \mid \bigoplus_{i\in I}\{L_i : \eta_i\} \mid \&\{L_i : \eta_i\}_{i\in(I_1,I_2)} \mid \psi$$

A session type is finished (end) or it can describe further interactions: the input ($?T.\eta$) or output ($!T.\eta$) of a *confined* value T, or the delegation ($!\eta'.\eta$) or resumption ($?\eta'.\eta$) of an endpoint of session type η', or the offering of non-deterministic selection ($\oplus\{L_i : \eta_i\}_{i\in I}$) of a label L_i, signifying that session type η_i is to be followed next.

Moreover, a session type can offer an external choice $\&\{L_i : \eta_i\}_{i\in(I_1,I_2)}$ to its communication partner. Here I_1 contains the labels that the process *must* be able to accept and I_2 the labels that it *may* accept. We require that I_1 and I_2 are disjoint and I_1 is not empty. Although a single set would suffice, the two sets make type inference deterministic and independent of source code order.

We express our session typing discipline as an *abstract interpretation semantics* for behaviours shown in Fig. 4. It describes transitions of the form $\Delta \vDash b \to_C \Delta' \vDash b'$, where b, b' are behaviours. The Δ and Δ' are stacks on which static endpoint labels together with their corresponding session types ($l : \eta$) can be pushed and popped. Inspired by Castagna et al. [3], in the transition $\Delta \vDash b \to_C \Delta' \vDash b'$, behaviour b can only use the top label in the stack to communicate, push another label on the stack, or pop the top label provided its session type is end. This stack principle gives us a partial lock freedom property (Thm. 5.2).

Rule End from Fig. 4 simply removes a finished stack frame, and rule Beta looks up behaviour variables in C; Plus chooses one of the branches of non-deterministic behaviour. The Push rule extends the stack by adding one more frame to it, as long as the label has not been added before on the stack (see Ex. 5.2). Rules Out and In reduce the top-level session type of the stack by an output and input, respectively. The requirement here is that the labels in the

$$
\begin{array}{lrll}
\textsc{End}: & (l:\mathsf{end})\cdot\Delta \vDash b & \to_C \Delta \vDash b & \\
\textsc{Beta}: & \Delta \vDash \beta & \to_C \Delta \vDash b & \text{if } C \vdash b \subseteq \beta \\
\textsc{Plus}: & \Delta \vDash b_1 \oplus b_2 & \to_C \Delta \vDash b_i & \text{if } i \in \{1,2\} \\
\textsc{Push}: & \Delta \vDash \mathsf{push}(l:\eta) & \to_C (l:\eta)\cdot\Delta \vDash \tau & \text{if } l \notin \Delta.\mathsf{labels} \\
\textsc{Out}: & (l:!T.\eta)\cdot\Delta \vDash \rho!T' & \to_C (l:\eta)\cdot\Delta \vDash \tau & \text{if } C \vdash \rho \sim l,\ T' <: T \\
\textsc{In}: & (l:?T.\eta)\cdot\Delta \vDash \rho?T' & \to_C (l:\eta)\cdot\Delta \vDash \tau & \text{if } C \vdash \rho \sim l,\ T <: T' \\
\textsc{Del}: & (l:!\eta_d.\eta)\cdot(l_d:\eta'_d)\cdot\Delta \vDash \rho!\rho_d & & \\
 & & \to_C (l:\eta)\cdot\Delta \vDash \tau & \text{if } C \vdash \rho \sim l,\ \rho_d \sim l_d,\ \eta'_d <: \eta_d \\
\textsc{Res}: & (l:?\eta_r.\eta) \vDash \rho?l_r & \to_C (l:\eta)\cdot(l_r:\eta_r) \vDash \tau & \text{if } (l \neq l_r),\ C \vdash \rho \sim l \\
\textsc{ICh}: & (l:\bigoplus_{i\in I}\{L_i:\eta_i\})\cdot\Delta \vDash \rho!L_j & \to_C (l:\eta_j)\cdot\Delta \vDash \tau & \text{if } (j \in I),\ C \vdash \rho \sim l \\
\textsc{ECh}: & (l:\&_{i\in(I_1,I_2)}\{L_i:\eta_i\})\cdot\Delta \vDash \&_{j\in J}\{\rho?L_j\,;b_j\} & & \\
 & & \to_C (l:\eta_k)\cdot\Delta \vDash b_k & \text{if } k \in J,\ C \vdash \rho \sim l, \\
 & & & I_1 \subseteq J \subseteq I_1 \cup I_2 \\
\textsc{Rec}: & \Delta \vDash \mathsf{rec}_\beta\, b & \to_C \Delta \vDash \tau & \text{if } \epsilon \vDash b \Downarrow_{C'}, \\
 & & & C' = (C \backslash (\mathsf{rec}_\beta\, b \subseteq \beta)) \cup (\tau \subseteq \beta) \\
\textsc{Spn}: & \Delta \vDash \mathsf{spawn}\, b & \to_C \Delta \vDash \tau & \text{if } \epsilon \vDash b \Downarrow_C \\
\textsc{Seq}: & \Delta \vDash b_1; b_2 & \to_C \Delta' \vDash b'_1; b_2 & \text{if } \Delta \vDash b_1 \to_C \Delta' \vDash b'_1 \\
\textsc{Tau}: & \Delta \vDash \tau\,; b & \to_C \Delta \vDash b &
\end{array}
$$

Fig. 4. Abstract Interpretation Semantics.

stack and the behaviour match, the usual subtyping [4] holds for the communicated types, and that the communicated types are *confined*. Note that sending confined (recursive) functions does not require delegation of endpoints.

Transfer of endpoints is done by delegate and resume (rules Del and Res). Delegate sends the second endpoint in the stack over the first; resume mimics this by adding a new endpoint label in the second position in the stack. Resume requires a one-frame stack to guarantee that the two endpoints of the same session do not end up in the same stack, thus avoiding deadlock [3]. If we abandon the partial lock freedom property guaranteed by our type system, then the conditions in Res can be relaxed and allow more than one frame.

A behaviour reduces an internal choice session type by selecting one of its labels (ICh). A behaviour offering an external choice is reduced non-deterministically to any of its branches (ECh). The behaviour must offer all *active* choices ($I_1 \subseteq J$) and all behaviour branches must be typable by the session type ($J \subseteq I_1 \cup I_2$).

As we previously explained, recursive functions in $\mathrm{ML_S}$ must be confined. This means that the communication effect of the function body is only on endpoints that the function opens internally, and the session type of these endpoints is followed to completion (or delegated) before the function returns or recurs. This is enforced in Rule Rec, where $\mathsf{rec}_\beta\, b$ must have no net effect on the stack, guaranteed by $\epsilon \vDash b \Downarrow_{C'}$. Here $C' = (C \backslash (\mathsf{rec}_\beta\, b \subseteq \beta)) \cup (\tau \subseteq \beta)$ is the original C with constraint $(\mathsf{rec}_\beta\, b \subseteq \beta)$ replaced by $(\tau \subseteq \beta)$ (cf., Def. 5.1). This update

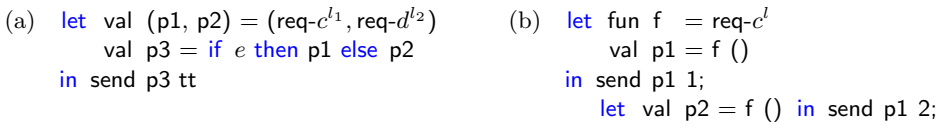

Fig. 5. Examples of aliasing

of C prevents the infinite unfolding of $\mathsf{rec}_b\,\beta$. Spawned processes must also be confined (Spn). We work with *well-formed* constraints:

Definition 5.1 (Well-Formed Constraints). *C is* well-formed *if:*

1. Type-Consistent: *for all type constructors tc_1, tc_2, if $(tc_1(\vec{t_1}) \subseteq tc_2(\vec{t_2})) \in C$, then $tc_1 = tc_2$, and for all $t_{1i} \in \vec{t_1}$ and $t_{2i} \in \vec{t_2}$, $(t_{1i} \subseteq t_{2i}) \in C$.*
2. Region-Consistent: *if $C \vdash l \sim l'$ then $l = l'$.*
3. Behaviour-Compact: *behaviour constraints cycles contain a $(\mathsf{rec}_\beta\, b \subseteq \beta) \in C$; also if $(\mathsf{rec}_\beta\, b \subseteq \beta') \in C$ then $\beta = \beta'$ and $\forall (b' \subseteq \beta) \in C$, $b' = \mathsf{rec}_\beta\, b$.*
4. Well-Confined: *if $C \vdash confd(T)$ then $T \neq \mathsf{Ses}^\rho$; also if $C \vdash confd(b)$ then $b \notin \{\rho!T, \rho?T, \rho!\rho, \rho?l, \rho!L_i, \underset{i\in I}{\&}\{\rho?L_i\,; b_i\}\}$.*

The first and fourth conditions are straightforward. The third condition disallows recursive behaviours through the environment without the use of a $\mathsf{rec}_\beta\, b$ effect. All well-typed $\mathrm{ML_S}$ programs contain only such recursive behaviours because recursion is only possible through the use of a recursive function. The second part of the condition requires that there is at most one recursive constraint in the environment using variable β. This is necessary for type preservation and decidability of session typing. The second condition of Def. 5.1 requires that only endpoints from a single source can flow in each ρ, preventing aliasing of endpoints generated at different source locations.

Example 5.1 (Aliasing of Different Sources). Consider the program in Fig. 5 (a). Which endpoint flows to p3 cannot be statically determined and therefore the program cannot yield a consistent session type for channels c and d. The program will be rejected in our framework because p3 has type Ses^ρ and from the constrain environment $C \vdash \rho \sim l_1,\ \rho \sim l_2$, which fails Def. 5.1.

Because endpoints generated from the same source code location are identified in our system, stacks are treated *linearly*: an endpoint label l may only once be pushed onto a stack. Every stack Δ contains an implicit set of the labels $\Delta.\mathsf{labels}$ to record previously pushed labels.

Example 5.2 (Aliasing From Same Source). Consider the program in Fig. 5 (b) where endpoint p1 has type Ses^ρ, with $C \vdash \rho \sim l$. The program has behaviour $\mathsf{push}(l : \eta); \rho!\mathsf{Int}; \mathsf{push}(l : \eta); \rho!\mathsf{Int}; \tau$. Label l is pushed on the stack twice and the behaviour complies with the session type $\eta = !\mathsf{Int}.\mathsf{end}$. However the program does not respect this session type because it sends two integers on p1 and none on p2. Our system rejects this program due to the violation of stack linearity.

Our system also rejects the correct version of the program in Fig. 5 (b), where the last send is replaced by $\mathsf{send\ p2\ 2}$. This is because the label l associated with the variable ρ of a type Ses^{ρ} is *control flow insensitive*. Existing techniques can make labels control flow sensitive (e.g., [14, 15]).

Using the semantics of Fig. 4 we define the following predicate which requires behaviours to follow to completion or delegate all $(l : \eta)$ frames in a stack.

Definition 5.2 (Strong normalization). *$\Delta \vDash b \Downarrow_C \vec{\Delta'}$ when for all b', Δ' such that $\Delta \vDash b \rightarrow_C^* \Delta' \vDash b' \not\rightarrow_C$ we have $b' = \tau$ and $\Delta' \in \{\vec{\Delta'}\}$. We write $\Delta \vDash b \Downarrow_C$ when $\Delta \vDash b \Downarrow_C \epsilon$, where ϵ is the empty stack.*

Lastly, session types on dual session endpoints ($c \sim \eta$, $\overline{c} \sim \eta'$) must be dual ($C \vdash \eta \bowtie \eta'$) The definition of duality is standard, with the exception that internal choice is dual to external choice only if the labels in the former are included in the *active* labels in the latter.

Definition 5.3 (Valid Constraint Environment). *C is* valid *if there exists a substitution σ of variables ψ with closed session types, such that $C\sigma$ is well-formed and for all $(c \sim \eta), (\overline{c} \sim \eta') \in C\sigma$ we have $C \vdash \eta \bowtie \eta'$.*

Combining the Two Levels

The key property here is *well-stackedness*, the fact that in a running system where each process has a corresponding stack of endpoints, there is a way to repeatedly remove pairs of endpoints with dual session types from the top of two stacks, until all stacks are empty.

Definition 5.4 (Well-stackedness). *$C \Vdash_{ws} \mathcal{S}$ is the least relation satisfying:*

$$\frac{}{C \Vdash_{ws} \epsilon} \qquad \frac{C \Vdash_{ws} \mathcal{S},\ \big(\Delta \vDash b,\ e\big),\ \big(\Delta' \vDash b',\ e'\big) \qquad C \vdash \eta \bowtie \eta' \qquad p, \overline{p} \sharp \Delta, \Delta', \mathcal{S}}{C \Vdash_{ws} \mathcal{S}, \big((p^l : \eta) \cdot \Delta \vDash b,\ e\big), \big((\overline{p}^{l'} : \eta') \cdot \Delta' \vDash b',\ e'\big)}$$

Note that this does not mean that programs are deterministic. Multiple pairs of endpoints may be at the top of a set of stacks. Duality of endpoints guarantees that communications are safe; the ordering of endpoints in removable pairs implies the absence of deadlocks.

We let P, Q range over tuples of the form $\big(\Delta \vDash b,\ e\big)$ and $\mathcal{S}$ over sequences of such tuples. In this section stack frames $(p^l : \eta)$ store both endpoints and their labels. We write $C \Vdash \overrightarrow{\big(\Delta \vDash b,\ e\big)}$ if C is well-formed and valid, $(C;\ \emptyset \vdash \overrightarrow{e : T \triangleright b})$, and $(\overrightarrow{\Delta \vDash b} \Downarrow_C)$, for some $\overrightarrow{T}$. We write $C \Vdash_{\mathsf{ws}} \mathcal{S}$ if $\overrightarrow{\Delta}$ is well-stacked. Well-typed systems enjoy session fidelity and preserve typing and well-stackedness.

Theorem 5.1. *Let $\mathcal{S} = \overrightarrow{\Delta \vDash b, e}$ and $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$ and $\overrightarrow{e} \longrightarrow \overrightarrow{e}'$; then there exist $\overrightarrow{\Delta}'$, $\overrightarrow{b}'$ such that $\mathcal{S}' = \overrightarrow{\big(\Delta' \vDash b',\ e'\big)}$ and:*

1. *$C \Vdash \mathcal{S}'$* *(Type Preservation)*
2. *$\overrightarrow{\Delta \vDash b} \rightarrow_C^* \overrightarrow{\Delta' \vDash b'}$* *(Session Fidelity)*
3. *$C \Vdash_{ws} \mathcal{S}'$* *(Well-Stackedness Preservation)*

Session fidelity and well-stackedness preservation imply *communication safety*, since the former guarantees that processes are faithful to session types in the stacks, while the latter that session types are dual for each pair of open endpoints p and $\bar{p}$. Moreover, well-stackedness implies *deadlock freedom*. P *depends on* Q if the endpoint at the top of P's stack has dual endpoint in Q.

Lemma 5.1 (Deadlock Freedom). *$C \Vdash_{ws} \mathcal{S}$; dependencies in $\mathcal{S}$ are acyclic.*

Type soundness is more technical. We divide system transitions to communication transitions between processes ($\longrightarrow_{\mathsf{c}}$) and internal transitions ($\longrightarrow_{\mathsf{i}}$). Let $\mathcal{S} \longrightarrow_{\mathsf{c}} \mathcal{S}'$ ($\mathcal{S} \longrightarrow_{\mathsf{i}} \mathcal{S}'$) when $S \longrightarrow S'$, derived by Rule RInit, RCom, RDel or RSel of Fig. 1 (resp., any other rule); $S \Longrightarrow_{\mathsf{c}} S'$ when $S \longrightarrow_{\mathsf{i}}^{*} \longrightarrow_{\mathsf{c}} \longrightarrow_{\mathsf{i}}^{*} S'$.

Theorem 5.2 (Type Soundness). *Let $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$. Then*

1. *$\mathcal{S} \Longrightarrow_{\mathsf{c}} \mathcal{S}'$, or*
2. *$\mathcal{S} \longrightarrow_{\mathsf{i}}^{*} (\mathcal{F}, \mathcal{D}, \mathcal{W}, \mathcal{B})$ such that:*
   Finished processes, $\mathcal{F}$: *$\forall P \in \mathcal{F}.\ P = \big(\epsilon \vDash \tau,\ v\big)$, for some v;*
   Diverging processes, $\mathcal{D}$: *$\forall P \in \mathcal{D}.\ P \longrightarrow_{\mathsf{i}}^{\infty}$;*
   Waiting proc., $\mathcal{W}$: *$\forall P \in \mathcal{W}.\ P = \big(\Delta \vDash b,\ E[e]\big)$ and $e \in \{\mathsf{req}\text{-}c^{l}, \mathsf{acc}\text{-}c^{l}\}$;*
   Blocked processes, $\mathcal{B}$: *$\forall P \in \mathcal{B}.\ P = \big(\Delta \vDash b,\ E[e]\big)$ and $e \in \{\mathsf{send}\, v, \mathsf{recv}\, v,$ $\mathsf{deleg}\ v, \mathsf{resume}\, v, \mathsf{sel}\text{-}L\, v, \mathsf{case}\, v\, \{L_i \Rightarrow e_i\}_{i \in I}\}$ and P transitively depends on a process in $\mathcal{D} \cup \mathcal{W}$.*

A well-typed and well-stacked $\mathrm{ML_S}$ system will either be able to perform a communication, or, after finite internal steps, it will reach a state where some processes are values ($\mathcal{F}$), some internally diverge ($\mathcal{D}$), some are waiting for a partner process to open a session ($\mathcal{W}$), and some are blocked on a session communication ($\mathcal{B}$). Crucially, in states where communication is not possible, $\mathcal{B}$ transitively depends on $\mathcal{D} \cup \mathcal{W}$. Thus, in the absence of divergence and in the presence of enough processes to start new sessions, no processes can be blocked; the system will either perform a communication or it will terminate *(partial lock freedom)*.

Corollary 5.1 (Partial Lock Freedom). *If $C \Vdash \mathcal{S}$, $C \Vdash_{ws} \mathcal{S}$, and $\mathcal{S} \not\Longrightarrow_{\mathsf{c}}$ and $\mathcal{S} \longrightarrow_{\mathsf{i}}^{*} (\mathcal{F}, \emptyset, \emptyset, \mathcal{B})$ then $\mathcal{B} = \emptyset$.*

6 Inference Algorithm

We use three inference algorithms, $\mathcal{W}$, $\mathcal{SI}$ and $\mathcal{D}$. The first infers functional types and communication effects and corresponds to the first level of our type system. The other two infer session types from the abstract interpretation rules of Fig. 4 ($\mathcal{SI}$) and the duality requirement of Def. 5.3 ($\mathcal{D}$), corresponding to the second level of the type system.

Algorithm $\mathcal{W}$ is a straightforward adaptation of the homonymous algorithm from [1]: given an expression e, $\mathcal{W}$ calculates its type t, behaviour b and constraints set C; no session information is calculated. $\mathcal{W}$ generates pairs of fresh constraints $c \sim \psi$ and $\overline{c} \sim \psi'$ for each global channel c in the source program; ψ and ψ' are unique. Results of $\mathcal{W}$'s soundness and completeness follow from [1].

For all constraints $(c \sim \psi) \in C$, Algorithm $\mathcal{SI}$ infers a substitution σ and a refined set C' such that $\epsilon \vDash b\sigma \Downarrow_{C'} \epsilon$. The substitution only maps ψ variables to session types. The final C' is derived from C by applying σ and possibly adding more type constraints of the form $(T \subseteq T')$. The core of this algorithm is the abstract interpreter $\mathcal{MC}$, which explores all possible transitions from $\epsilon \vDash b$.

Algorithm $\mathcal{MC}$ is designed in a continuation-passing style, using a *continuation stack* $K ::= \epsilon \mid b \cdot K$.

As transition paths are explored, previously discovered branches of internal and external choices in session types may need to be expanded. For example, if Algorithm $\mathcal{MC}$ encounters a configuration $(l : \oplus\{L_i : \eta_i\}_{i \in I}) \vDash l!L_j$ where $j \notin I$, the inference algorithm needs to add the newly discovered label L_j to the internal choice on the stack.

To do this, internal and external choices are removed from the syntax of sessions, and replaced with special variables ψ_{in} and ψ_{ex}. These variables are bound by unique *choice constraints*, extending the syntax of constraints (Fig. 2):

$$C ::= \ldots \mid \oplus\{L_i : \eta_i\}_{i \in I} \sim \psi_{\mathsf{in}} \mid \&\{L_i : \eta_i\}_{i \in (I_1, I_2)} \sim \psi_{\mathsf{ex}}$$

$\mathcal{MC}$ updates ψ_{in} and ψ_{ex} constraints in C with newly discovered branches. For example it may add new labels to an internal choice, or move active labels to inactive in an external choice.

We now give more detail for some inference steps of Algorithm $\mathcal{MC}$. The full algorithm can be found in an online technical report[1]. Algorithm $\mathcal{MC}$ terminates successfully when all sessions on the stack have terminated, the input behaviour is τ and the continuation stack is empty:

$$\mathcal{MC}(\Delta \vDash \tau,\ C,\ \epsilon) = (\sigma,\ C\sigma)$$
$$\text{if}\ \ \sigma = \mathsf{finalize}\ \Delta$$

When this clause succeeds, Δ may be empty or it may contain frames of the form $(l : \psi)$ or $(l : \mathsf{end})$. The helper function $\mathsf{finalise}\ \Delta$ returns a substitution σ that maps all such ψ's to end. If this is not possible (i.e., a session on Δ is not finished) $\mathsf{finalise}$ raises an error.

New frames are pushed on the stack when the behaviour is $\mathsf{push}(l : \eta)$:

$$\mathcal{MC}(\Delta \vDash \mathsf{push}(l : \eta),\ C,\ K) = (\sigma_2\sigma_1, C_2)$$
$$\text{if}\ \ (\sigma_1, \Delta_1) = \mathsf{checkFresh}(l, \Delta)$$
$$\text{and}\ (\sigma_2, C_2) = \mathcal{MC}((l : \eta\sigma_1) \cdot \Delta_1 \vDash \tau,\ C\sigma_1,\ K\sigma_1)$$

where $\mathsf{checkFresh}$ checks that l has never been in Δ.

When the behaviour is an operation that pops a session from the stack, such as a send $(l!T)$, $\mathcal{MC}$ looks up the top frame on the stack, according to the stack principle. There are two cases to consider: either the top frame contains a fresh variable ψ, or some type has been already inferred. The algorithm here is:

[1] Spaccasassi, C., Koutavas, V.: Type-Based Analysis for Session Inference. ArXiv e-prints (Oct 2015), `http://arxiv.org/abs/1510.03929v3`.

$$\mathcal{MC}\big((l:\psi)\cdot\Delta \vDash \rho!T,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C \vdash l \sim \rho$$
$$\quad \text{and } \sigma_1 = [\psi \mapsto\ !\alpha.\psi'] \text{ where } \alpha, \psi' \text{ fresh}$$
$$\quad \text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l:\psi')\cdot\Delta\sigma_1 \vDash \tau,\ C\sigma_1 \cup \{T \subseteq \alpha\},\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l:!\alpha.\eta)\cdot\Delta \vDash \rho!T,\ C,\ K\big) = \mathcal{MC}\big((l:\eta)\cdot\Delta \vDash \tau,\ C \cup \{T \subseteq \alpha\},\ K\big)$$
$$\quad \text{if} \quad C \vdash l \sim \rho$$

In the first case, $\mathcal{MC}$ checks that ρ in the behaviour corresponds to l at the top of the stack. It then produces the substitution $[\psi \mapsto\ !\alpha.\psi']$, where α and ψ' are fresh, and adds $(T \subseteq \alpha)$ to C. The second case produces no substitution.

The clauses for delegation are similar:

$$\mathcal{MC}\big((l:\psi)\cdot(l_d:\eta_d)\cdot\Delta \vDash \rho!\rho_d,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if } \ C \vdash l \sim \rho \text{ and } C \vdash l_d \sim \rho_d$$
$$\quad \text{and } \sigma_1 = [\psi \mapsto\ !\eta_d.\psi'] \text{ where } \psi' \text{ fresh}$$
$$\quad \text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l:\psi')\cdot\Delta\sigma \vDash \tau,\ C\sigma_1,\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l:!\eta_d.\eta)\cdot(l_d:\eta'_d)\cdot\Delta \vDash \rho!\rho_d,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C \vdash l \sim \rho \text{ and } C \vdash l_d \sim \rho_d$$
$$\quad \text{and } (\sigma_1, C_1) = \ \mathsf{sub}(\eta'_d, \eta_d, C)$$
$$\quad \text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l:\eta)\cdot\Delta\sigma \vDash \tau,\ C_1,\ K\sigma_1\big)$$

The main difference here is that, in the second clause, the sub function checks that $C \vdash \eta'_d <: \eta_d$ and performs relevant inference. Moreover, the input Δ must contain at least two frames (the frame below the top one is delegated).

The cases for receive, label selection and offer, and resume are similar (see online report). In the cases for label selection and offering, the algorithm updates the ψ_{in} and ψ_{ex} variables, as discussed above. In the case of resume, the algorithm checks that the stack contains one frame.

In behaviour sequencing and branching, substitutions are applied eagerly and composed iteratively, and new constraints are accumulated in C:

$$\mathcal{MC}\big(\Delta \vDash b_1\,;b_2,\ C,\ K\big) = \mathcal{MC}\big(\Delta \vDash b_1,\ C,\ b_2 \cdot K\big)$$

$$\mathcal{MC}\big(\Delta \vDash b_1 \oplus b_2,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad (\sigma_1, C_1) = \mathcal{MC}\big(\Delta \vDash b_1,\ C,\ K\big)$$
$$\quad \text{and } (\sigma_2, C_2) = \mathcal{MC}\big(\Delta\sigma_1 \vDash b_2\sigma_1,\ C_1,\ K\sigma_1\big)$$

When a recursive behaviour $\mathsf{rec}_b\,\beta$ is encountered, Algorithm $\mathcal{MC}$ needs to properly setup the input constraints C according to Rule Rec of Fig. 4:

$$\mathcal{MC}(\Delta \vDash \mathsf{rec}_\beta\, b,\ C) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C = C' \uplus \{b' \subseteq \beta\}$$
$$\quad \text{and } (\sigma_1, C_1) = \mathcal{MC}\big(\epsilon \vDash b,\ C' \cup \{\tau \subseteq \beta\},\ \epsilon\big)$$
$$\quad \text{and } (\sigma_2, C_2) = \mathcal{MC}\big(\Delta\sigma_1 \vDash \tau,\ (C_1 \backslash \{\tau \subseteq \beta\}) \cup (\{b' \subseteq \beta\})\sigma_1,\ K\sigma_1\big)$$

Here the algorithm first calls $\mathcal{MC}$ on $\epsilon \vDash b$, checking that the recursion body b is self-contained under C', in which the recursion variable β is bound to τ. This update of C prevents the infinite unfolding of $\mathsf{rec}_b\,\beta$. It then restores back the constraint on β, applies the substitution σ_1, and continues inference.

The clause for $\mathsf{spawn}\, b$ is similar, except that C is unchanged. Variables β are treated as the internal choice of all behaviours b_i bound to β in C:

$$\mathcal{MC}(\Delta \vDash \beta,\ C,\ K) = \mathcal{MC}(\Delta \vDash b,\ C,\ K)$$
$$\text{where}\quad b = \bigoplus\{b_i \mid \exists i.\ (b_i \subseteq \beta) \in C\}$$

Inference fails when $\mathcal{MC}$ reaches a stuck configuration $\Delta \vDash b$ other than $\epsilon \vDash \tau$, corresponding to an error in the session type discipline.

To prove termination of $\mathcal{SI}$, we first define the translation $[\![b]\!]^{\mathsf{g}}_C$, that replaces β variables in b with the internal choice $\bigoplus\{\, b_i \,|\, (\{b_i \subseteq \beta\}) \in C \,\}$. Due to behaviour-compactness (Def. 5.1), $[\![b]\!]^{\mathsf{g}}_C$ is a finite *ground* term, i.e. a finite term without β variables. Except for Rule Beta, transitions in Fig. 4 never expand b; they either consume Δ or b. Since $[\![b]\!]^{\mathsf{g}}_C$ is finite when C is well-formed, $\epsilon \vDash [\![b]\!]^{\mathsf{g}}_C$ generates a finite state space and Algorithm $\mathcal{MC}$ always terminates.

Similar to ML type inference, the worst-case complexity of $\mathcal{MC}$ is exponential to program size: $\mathcal{MC}$ runs in time linear to the size of $[\![b]\!]^{\mathsf{g}}_C$, which in the worst case is exponentially larger than b, which is linear to program size. The worst case appears in pathological programs where, e.g., each function calls all previously defined functions. We intend to explore whether this is an issue in practice, especially with an optimised dynamic programming implementation of $\mathcal{MC}$.

Soundness and completeness of $\mathcal{SI}$ follow from the these properties of $\mathcal{MC}$.

Lemma 6.1 (Soundness of $\mathcal{MC}$). *Let C be well-formed and $\mathcal{MC}(\Delta \vDash b, C) = (\sigma_1, C_1)$; then $\Delta\sigma_1 = \Delta'$ and $\Delta' \vDash b\sigma_1 \Downarrow_{C_1}$.*

Lemma 6.2 (Completeness of $\mathcal{MC}$). *Let C be well-formed and $(\Delta \vDash b)\sigma \Downarrow_C$; then $\mathcal{MC}(\Delta \vDash b, C_0) = (\sigma_1, C_1)$ and $\exists\sigma'$ such that $C \vdash C_1\sigma'$ and $\forall\psi \in dom(\sigma)$, $C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$.*

Completeness states that $\mathcal{MC}$ computes the most general constraints C_1 and substitution σ_1, because, for any C and σ such that $(\Delta \vDash b)\sigma$ type checks, C specialises C_1 and σ is an instance of σ_1, after some extra substitution σ' of variables (immaterial for type checking).

Algorithm $\mathcal{D}$ collects all $c \sim \eta_1$ and $\bar{c} \sim \eta_2$ constraints in C', generates duality constraints $\eta_1 \bowtie \eta_2$ and iteratively checks them, possibly substituting ψ variables. It ultimately returns a C'' which is a valid type solution according to Def. 5.3. Soundness and completeness of Algorithm $\mathcal{D}$ is straightforward.

We now show how $\mathcal{SI}$ infers the correct session types for Ex. 2.1 from Sect. 2. We assume that Algorithm $\mathcal{W}$ has already produced a behaviour b and constraints C for this example. For clarity, we simplify b and C: we remove spurious τs from behaviour sequences, replace region variables ρ with labels (only one label flows to each ρ), and perform simple substitutions of β variables.

Example 6.1 (A Swap Service).

There are three textual sources of endpoints in this example: the two occurrences of acc-swp in coord, and req-swp in swap. A pre-processing step automatically annotates them with three unique labels l_1, l_2 and l_3. Algorithm $\mathcal{W}$ infers b and C for Ex. 2.1; the behaviour b (simplified) is:

$$\mathsf{spawn}\,(\beta_{coord}); \mathsf{spawn}\,(\beta_{swap}); \mathsf{spawn}\,(\beta_{swap})$$

In this behaviour three processes are spawned: one with a β_{coord} behaviour, and two with a β_{swap} behaviour. The behaviour associated to each of these variables is described in C, along with other constraints:

1. $\mathsf{rec}_{\beta_{coord}}\,(\mathsf{push}(l_1 : \psi_1); l_1?\alpha_1; \mathsf{push}(l_2 : \psi_1); l_2?\alpha_2; l_2!\alpha_1; l_1!\alpha_2); \beta_{coord} \subseteq \beta_{coord}$
2. $\mathsf{push}(l_3 : \psi_2); l_3!\mathsf{Int}; l_3?\alpha_3 \subseteq \beta_{swap}$
3. $\overline{swap} \sim \psi_1$
4. $swap \sim \psi_2$

The above behaviour and environment are the inputs to Algorithm $\mathcal{SI}$, implementing session type inference according to the second level of our framework. The invocation $\mathcal{SI}(b, C)$ calls $\mathcal{MC}\big(\epsilon \vDash b,\ C,\ \epsilon\big)$, where the first ϵ is the empty endpoint stack Δ and the second ϵ is the empty continuation stack. Behaviour b is decomposed as $b = K[b']$, where $b' = \mathsf{spawn}\,(\beta_{coord})$ and K is the continuation $[\ ]; \mathsf{spawn}\,(\beta_{swap}); \mathsf{spawn}\,(\beta_{swap})$. The algorithm thus calls $\mathcal{MC}\big(\epsilon \vDash \mathsf{spawn}\,(\beta_{coord}),\ C,\ K\big)$, which, after replacing β_{coord} and unfolding its inner recursive behaviour becomes:

$$\mathcal{MC}\big(\epsilon \vDash \mathsf{push}(l_1 : \psi_1); l_1?\alpha_1; \mathsf{push}(l_2 : \psi_1); l_2?\alpha_2; l_2!\alpha_1; l_1!\alpha_2; \beta_{coord},\ C_1,\ \epsilon\big)$$

Here C_1 is equal to C above, with the exception of replacing Constraint 1 with the constraint $(\tau \subseteq \beta_{coord})$. Inference is now straightforward: the frame $(l_1 : \psi_1)$ is first pushed on the endpoint stack. From behaviour $l_1?\alpha_1$ the algorithm applies substitution $[\psi_1 \mapsto ?\alpha_4.\psi_4]$, where ψ_4 and α_4 are fresh, and generates constraint $(\alpha_4 \subseteq \alpha_1)$ obtaining C_2. We thus get:

$$\mathcal{MC}\big((l_1 : \psi_4) \vDash \mathsf{push}(l_2 : ?\alpha_4.\psi_4); l_2?\alpha_2; l_2!\alpha_1; l_1!\alpha_2; \beta_{coord},\ C_2,\ \epsilon\big)$$

After the next push, the endpoint stack becomes $(l_2 : ?\alpha_4.\psi_4) \cdot (l_1 : \psi_4)$. The next behaviour $l_2?\alpha_2$ causes $\mathcal{MC}$ to create constraint $(\alpha_4 \subseteq \alpha_2)$ obtaining C_3, and to consume session $?\alpha_4$ from the top frame of the endpoint stack.

$$\mathcal{MC}\big((l_2 : \psi_4) \cdot (l_1 : \psi_4) \vDash l_2!\alpha_1; l_1!\alpha_2; \beta_{coord},\ C_3,\ \epsilon\big)$$

Because of $l_2!\alpha_1$, $\mathcal{MC}$ generates $[\psi_3 \mapsto\ !\alpha_5.\psi_5]$ and $(\alpha_1 \subseteq \alpha_5)$ obtaining C_4.

$$\mathcal{MC}\big((l_2 : \psi_5) \cdot (l_1 : !\alpha_5.\psi_5) \vDash l_1!\alpha_2; \beta_{coord},\ C_4,\ \epsilon\big)$$

Since l_1 in the behaviour and l_2 at the top of the endpoint stack do not match, $\mathcal{MC}$ infers that ψ_5 must be the terminated session end. Therefore it substitutes $[\psi_5 \mapsto \mathsf{end}]$ obtaining C_5. Because of the substitutions, C_5 contains $\overline{swap} \sim ?\alpha_4.!\alpha_5.\mathsf{end}$. After analysing β_{swap}, $\mathcal{MC}$ produces C_6 where $swap \sim !\mathsf{Int}.!\alpha_6.\mathsf{end}$.

During the above execution $\mathcal{MC}$ verifies that the stack principle is respected and no endpoint label is pushed on the stack twice. Finally the algorithm calls $\mathcal{D}(C_6)$ which performs a duality check between the constraints of $\overline{swap}$ and $swap$, inferring substitution $[\alpha_4 \mapsto \mathsf{Int},\ \alpha_6 \mapsto \alpha_5]$. The accumulated constraints on type variables α give the resulting session types of the swap channel endpoints: $(\overline{swap} \sim ?\mathsf{Int}.!\mathsf{Int}.\mathsf{end})$ and $(swap \sim !\mathsf{Int}.?\mathsf{Int}.\mathsf{end})$.

7 A Proposal for Recursive Session Types

The system we have presented does not include recursive session types. Here we propose an extension to the type system with recursive types. The inference algorithm for this extension is non-trivial and we leave it to future work.

In this extension, a recursive behaviour may partially use a recursive session type and rely on the continuation behaviour to fully consume it. First we add *guarded* recursive session types: $\eta ::= \ldots \mid \mu X.\eta \mid X$. The first level of our type system remains unchanged, as it is parametric to session types, and already contains recursive functions and behaviours.

A recursive behaviour $\mathsf{rec}_\beta\, b$ operating on an endpoint l with session type $\mu X.\eta$ may: (a) run in an infinite loop, always unfolding the session type; (b) terminate leaving l at type end; (c) terminate leaving l at type $\mu X.\eta$. Behaviour b may have multiple execution paths, some terminating, ending at τ, and some recursive, ending at a recursive call β. They all need to leave l at the same type, either end or $\mu X.\eta$; the terminating paths of b determine which of the two session types l will have after $\mathsf{rec}_\beta\, b$. If b contains no terminating paths then we assume that l is fully consumed by $\mathsf{rec}_\beta\, b$ and type the continuation with l at end.

To achieve this, we add a *stack environment* D in the rules of Fig. 4, which maps labels l to stacks Δ. If $\Delta_1 = (l : \mu X.\eta)$, we call an *$l$-path* from $\Delta_1 \vDash b_1$ any finite sequence of transitions such that $\Delta_1 \vDash b_1 \to_{C,D} \ldots \to_{C,D} \Delta_n \vDash b_n \not\to_{C,D}$. A l-path is called *l-finitary* if there is no $b_i = \tau^l$ for any configuration i in the series; otherwise we say that the path is *l-recursive*. We write $(l : \mu X.\eta) \vDash b \Downarrow^{\mathsf{fin}}_{C,D} \Delta'$ when the last configuration of all l-finitary paths from $(l : \mu X.\eta) \vDash b$ is $\Delta' \vDash \tau$. Similarly, we write $(l : \mu X.\eta) \vDash b \Downarrow^{\mathsf{rec}}_{C,D} \Delta'$ when the last configuration of all l-recursive paths from $(l : \mu X.\eta) \vDash b$ is $\Delta' \vDash \tau^l$. When no $l-$ paths from $(l : \mu X.\eta) \vDash b$ is l-finitary, we stipulate $(l : \mu X.\eta) \vDash b \Downarrow^{\mathsf{fin}}_{C,D} (l : \mathsf{end})$ holds. We add the following rules to those of Fig. 4.

Rec2

$$\frac{(l : \mu X.\eta) \vDash b \Downarrow^{\mathsf{fin}}_{C',D'} \Delta' \qquad (l : \mu X.\eta) \vDash b \Downarrow^{\mathsf{rec}}_{C',D'} \Delta'}{(l : \mu X.\eta) \cdot \Delta \vDash \mathsf{rec}_\beta\, b \to_{C,D} \Delta' \cdot \Delta \vDash b'}$$

$\Delta' \in \{(l : \mathsf{end}), (l : \mu X.\eta)\}$
$C' = (C \backslash (\mathsf{rec}_\beta\, b \subseteq \beta)) \cup (\tau^l \subseteq \beta)$
$D' = D[l \mapsto \Delta']$

RCall

$$\frac{}{(l : \mu X.\eta) \vDash \tau^l \to_{C,D} D(l) \vDash \tau}$$

Unf

$$\frac{(l : \eta[X \mapsto \mu X.\eta]) \cdot \Delta \vDash b \to_{C,D} \Delta' \vDash b'}{(l : \mu X.\eta) \cdot \Delta \vDash b \to_{C,D} \Delta' \vDash b'}$$

Rule Rec2 requires that both l-finitary and l-recursive paths converge to the same stack Δ', either $(l : \mathsf{end})$ or $(l : \mu X.\eta)$. In this rule, similarly to rule Rec in Fig. 4, we replace the recursive constraint $(\mathsf{rec}_\beta\, b \subseteq \beta)$ with $(\tau^l \subseteq \beta)$, representing a trivial recursive call of β. This guarantees that all l-paths have a finite number of states. The D environment is extended with $l \mapsto \Delta'$, used in Rule RCall to obtain the session type of l after a recursive call. Rule Unf simply unfolds a recursive session type.

8 Related Work and Conclusions

We presented a new approach for adding binary session types to high-level programming languages, and applied it to a core of ML with session communication. In the extended language our system checks the session protocols of interesting programs, including one where pure code calls library code with communication effects, without having to refactor the pure code (Ex. 2.3). Type soundness guarantees partial lock freedom, session fidelity and communication safety.

Our approach is modular, organised in two levels, the first focusing on the type system of the source language and second on typing sessions; the two levels communicate through *effects*. In the fist level we adapted and extended the work of Amtoft, Nielson and Nielson [1] to session communication, and used it to extract the communication effect of programs. In the second level we developed a session typing discipline inspired by Castagna et al. [3]. This modular approach achieves a provably complete session inference for finite sessions without programmer annotations.

Another approach to checking session types in high-level languages is to use substructural type systems. For example, Vasconcelos et al. [18] develop such a system for a functional language with threads, and Wadler [19] presents a linear functional language with effects. Type soundness in the former guarantees session fidelity and communication safety, and in the latter also lock freedom and strong normalisation. Our system is in between these two extremes: lock freedom is guaranteed only when processes do not diverge and their requests for new sessions are met. Other systems give similar guarantees (e.g., [3, 16]).

Toninho et al. [16] add session-typed communication to a functional language using a monad. Monads, similar to effects, cleanly separate session communication from the rest of the language features which, unlike effects, require parts of the program to be written in a monadic style. Pucella and Tov [13] use an indexed monad to embed session types in Haskell, however with limited endpoint delegation: delegation relies on moving capabilities, which cannot escape their static scope. Our Ex. 2.2 is not typable in that system because of this. In [13] session types are inferred by Haskell's type inference. However, the programmer must guide inference with expressions solely used to manipulate type structures.

Tov [17] has shown that session types can be encoded in a language with a general-purpose substructural type system. Type inference alleviates the need for typing annotations in the examples considered. Completeness of session inference relies on completeness of inference in the general language, which is not clear.

Igarashi et al. [9] propose a reconstruction algorithm for finite types in the linear π calculus. Inference is complete and requires no annotations. Padovani [11] extends this work to pairs, disjoint sums and regular recursive types.

Mezzina [10] gives an inference algorithm for session types in a calculus of services. The type system does not support recursive session types and endpoint delegation. It does allow, however to type replicated processes that only use finite session types, similar to our approach.

Bibliography

[1] Amtoft, T., Nielson, H.R., Nielson, F.: Type and effect systems - behaviours for concurrency. Imperial College Press (1999)

[2] Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR, LNCS, vol. 6269, pp. 222–236. Springer (2010)

[3] Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of session types. In: PPDP. pp. 219–230. ACM (2009)

[4] Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica 42(2-3), 191–225 (2005)

[5] Gay, S., Vasconcelos, V.: Linear type theory for asynchronous session types. Journal of Functional Programming 20(01), 19–50 (Jan 2010)

[6] Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR, LNCS, vol. 715, pp. 509–523. Springer (1993)

[7] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) PLS (ESOP), LNCS, vol. 1381, pp. 122–138. Springer (1998)

[8] Hüttel, H., Lanese, I., Vasconcelos, V., Caires, L., Carbone, M., Deniélou, P., Padovani, L., Ravara, A., Tuosto, E., Vieira, H., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comp. Surv. To appear

[9] Igarashi, A., Kobayashi, N.: Type reconstruction for linear π-calculus with I/O subtyping. Inf. Comput. 161(1), 1–44 (2000)

[10] Mezzina, L.G.: How to infer finite session types in a calculus of services and sessions. In: Lea, D., Zavattaro, G. (eds.) Coordination Models and Languages, LNCS, vol. 5052, pp. 216–231. Springer (2008)

[11] Padovani, L.: Type reconstruction for the linear π-calculus with composite regular types. Logical Methods in Computer Science 11(4) (2015)

[12] Palsberg, J.: Type-based analysis and applications. In: PASTE. pp. 20–27. ACM (2001)

[13] Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Haskell Symposium. pp. 25–36. ACM (2008)

[14] Shivers, O.: Control-flow analysis of higher-order languages. Ph.D. thesis, CMU (1991)

[15] Tofte, M., Talpin, J.: Implementation of the typed call-by-value lambda-calculus using a stack of regions. In: POPL. pp. 188–201. ACM (1994)

[16] Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: Felleisen, M., Gardner, P. (eds.) PLS (ESOP), LNCS, vol. 7792, pp. 350–369. Springer (2013)

[17] Tov, J.: Practical Programming with Substructural Types. Ph.D. thesis, Northeastern University (2012)

[18] Vasconcelos, V., Gay, S., Ravara, A.: Type checking a multithreaded functional language with session types. Th. Computer Sc. 368(1-2), 64–87 (2006)

[19] Wadler, P.: Propositions as sessions. In: ICFP. pp. 273–286. ACM (2012)