Session Types with Gradual Typing

Peter Thiemann

University of Freiburg, Georges-Khler-Allee 079, 79110 Freiburg, Germany,
thiemann@acm.org

Abstract. Session types enable fine-grained static control over communication protocols. Gradual typing is a means to safely integrate statically and dynamically typed program fragments.
We propose a calculus for synchronous functional two-party session types, augment this calculus with a dynamically typed fragment as well as coercion operations between statically and dynamically typed parts, and establish its basic metatheory: type preservation and progress. A technical novelty is the notion of coercions for the choice operator in session types which is related to coercions of sum types.

Keywords: session types, gradual typing, coercion calculus

1 Introduction

Session types enable fine-grained static control over communication protocols. They evolved from a structuring scheme for two-party communication in π-calculus [11] over calculi embedded in functional languages [6, 7, 19] to a powerful means of describing multi-party orchestration of communication [13]. There are various embeddings in object-oriented languages [2, 8] as well as uses in the context of scripting languages [12, 14]. Their logical foundations have been investigated with interpretations in intuitionistic and classical linear logic [1, 20].

Gradual types [16], on the other hand, were conceived to improve the efficiency of dynamically typed programs by imposing static typing where possible and by resorting to dynamic checking where necessary. However, gradual types have further applications. For example, the blame calculus [21] explores the safe interaction of statically and dynamically typed program fragments on the basis of gradual typing. There are also gradual versions of type systems for information flow [3, 5] and typestate [22].

Our work explores the safe interplay of statically and dynamically typed program fragments in the context of session types. We start from a session-typed functional calculus with synchronous communication and augment it with a type D (dynamic) along with coercions into and out of that type. The resulting calculus has interesting applications inspired by recent work on applying session types in connection with scripting languages [12, 14]. In particular, in the cited work, program stubs in a dynamically typed language are generated for clients (and servers) from a session type specification. Our calculus can be used for

dynamically ensuring type safety when such a stub is used in connection with a statically typed server (client).

In session types, the choice operations on the sender and receiver side naturally generate a subtyping relation [6]: a receiver may accept more alternatives than provided by a sender. Our calculus comes with a new, dynamically checked notion of sender and receiver choice coercions, which allow a receiver to accept *fewer* alternatives than provided by the sender. These coercions are useful to manage prototyping situations, where a communication peer initially implements only parts of a protocol or where the peer implements a protocol extension before it becomes part of the agreed upon protocol specification. Choice coercions are closely related to gradual typing for sum types, which has not be investigated in the literature so far.

Our calculus allows the application of coercions to all communications in a channel, but it does not allow the coercion of an entire session type to type D. We leave this additional step to future work because it requires a closer look at the interaction between dynamic types and linear/affine types as investigated by prior work [4, 18, 22]. In any case, our calculus seems to be a good fit for the intended scenario of ensuring type safety for programs that are elaborations of generated program stubs. In the dynamic part, the generated code guarantees adherence to the message sequencing constraints imposed by the session typing. Our calculus guarantees the type checking for the transmitted values on both ends of the communication.

Furthermore, the work on gradual types for improving the efficiency of dynamically typed programs [16] imposes a type discipline on an underlying untyped calculus and then specifies a type-driven transformation that introduces coercions. Our calculus follows the approach of the blame calculus [21] and considers what can be seen as the target language of the transformation after coercion introduction. This latter choice is appropriate when considering programs composed of statically and dynamically typed fragments where the programmer uses explicit coercions to demarcate the fragments.

Overview. Sec. 2 introduces the session calculus with gradual types with examples. Sec. 3 contains the syntax, semantics, and typing of the calculus. Sec. 4 proves type soundness. Sec. 5 discusses related work.

2 Sessions Between Static and Dynamic Peers

Consider a client-server system that implements the web interface of a weather forecast system. For this system, a simple protocol between client and server might look like this: The client first sends its location as a number (postcode or GPS coordinate, type `num`). Next, it may either request the current temperature measurement (m) or a comprehensive report including historic data and forecasts (r). After sending an m-request the client expects the current measurement as a number. After sending an r-request the client expects a `num->num` function that returns measurements according to its input. Zero corresponds to

```
client(p, location, single) {
  cl = request p
  cs = send (location, cl)
  if (single) { ca = select m cs
                (n:num, ca') = receive ca
                displayCurrent (n)
                close ca'
  } else {      cb = select r cs
                (f:num->num, cb') = receive cb
                display14DayAverage (f)
                close cb'
}}
```

Fig. 1. Statically typed client code.

present, negative arguments return past measurements, positive arguments yield forecasts. Results are meaningful only in a certain, undetermined range.[1]

This protocol is expressed as a session type on the port p providing the service with $\oplus\langle\dots\rangle$ denoting a sender choice between sessions starting with label `m` or `r`, $!t.s$ ($?t.s$) sending (receiving) an item of type t and continuing with s, and `END` indicating the end of the connection:

$$p : [!\,\mathtt{num}.\oplus\langle \mathtt{m} : ?\,\mathtt{num}.\mathtt{END}, \mathtt{r} : ?\,(\mathtt{num} \to \mathtt{num}).\mathtt{END}\rangle]$$

Figure 1 contains code for statically typed client. It performs a request on the port and obtains a channel typed with the session type associated to the port. It then runs the protocol on the channel using the usual primitives of sessions types: `send` and `receive` for simple messages, `select x c` for selecting alternative `x` on connection `c`, and `close` for closing a connection. The assignment notation in the code abbreviates sequences of let-expressions.

At the other end, the server may be written in a dynamically typed language (Figure 2).[2] Although the server can accept a connection on port `p`, it cannot directly communicate on the connection obtained from it because the channel does not accept dynamic values. For that reason, the server first coerces the connection to process dynamically typed values.

The session type of the accepted connection is the dual of the client's session type: $?\,\mathtt{num}.\&\langle \mathtt{m} : !\,\mathtt{num}.\mathtt{END}, \mathtt{r} : !\,(\mathtt{num} \to \mathtt{num}).\mathtt{END}\rangle$. The cast applied to it is expressed using a coercion calculus with # standing for coercion application and ; for left-to-right sequential composition of coercions. For each (session-) type constructor, there is a corresponding coercion constructor that takes a coercion that is applied to each constituent type of the constructor. The example code uses the coercion operator for receiving choice, which fits to the type of the accepted connection. The `m` compartment of the coercion contains a coercion

[1] The protocol does not transmit the range to keep it simple.

[2] The code makes all coercions explicit, including those that need not be written by the programmer.

```
server(p) {
  cl = accept p #
    ?(num↑).&⟨m : !(num↓^{b.1}).END,  r : !((D → D)↓^{b.2} ;(num↑→ num↓^{b.3})).END⟩
  (location, cd) = receive cl
  case cd of {
  m : fun ca => ca' = send ( 24.3 #  num↑, ca)
                close ca'
  r : fun cb => cb' = send (
          fun z => (24.3 + (z # num↓^b)*0.1) # num↑ #(D → D)↑, cb)
                close cb'
  }
}
```

Fig. 2. Dynamically typed server code.

for the value before it is sent to the client: $\texttt{num}\downarrow^{b.1}$ checks that a value of type dynamic is indeed a `num` and extracts it. The annotation $b.1$ is a *blame label* that identifies the source of the (potential) run-time error when the types do not match. The coercion in the `r` compartment first checks that its argument is a function and then applies a function coercion that transforms that argument and result type to the desired target types. Assuming a code generator for session types, all of this code except the boxed fragments would be generated.

2.1 Coercions as Proxies

Applying the coercion to the accepted session creates a proxy process that mediates between the existing connection and the new connection. This proxy maintains a new channel of the desired target type of the coercion. The proxy forwards the operations between the channels as prescribed by the session types. It applies the cast operations from the session coercion before writing to the other end of a channel. It also retains the blame labels. Conceptually, the construction of the proxy could be done by a program transformation before execution as demonstrated with the transformed server program in Fig. 3.

2.2 Choice Coercions

There are two kinds of choice operators in session types, the internal choice $\oplus\langle\ldots\rangle$ where the program selects a particular outcome and the external choice where the program reacts and makes a case distinction driven by the label it receives. The coercion operator corresponding to choice supplies a coercion for each case mentioned in the type.

As a novel feature, a coercion may also add or remove cases from the set of labels on which the session type branches. This flexibility is achieved by only restricting the argument type of a coercion to match on the common branch labels. The result type of the coercion specifies the branches in the coerced session type. The branches which are not specified by the coercion result in

```
server_eager(p) {
  cs' = accept p
  p' = new port [?num.&⟨m : ! D.END, r : ! D.END⟩]
  cl = pipe p' (proxy cs') (fun cl => cl)
  (loc, cd) = receive cl
  ...
}
proxy = fun cl cl' {
  (dl : D, cp) = receive cl
  cp' = send (dl # num↑, cl')
  case cp of {
  m : fun ca => (df : D, ca1) = receive ca
                ca2 = send (df # num↓^{b.1}, ca1)
                close ca2
  r : fun cb => (df : D, cb1) = receive cb
                cb2 = send (df # (D → D)↓^{b.2} ;(num↑→ num↓^{b.3}), cb1)
                close cb2
}}
```

Fig. 3. Transformed server with proxy implementation of session coercion.

```
partial_server(p) {
  cl = accept p # ?(num↑).&^{r↦b.0}_{{m,r}}⟨m : !(num↓^{b.1}).END⟩
  (loc, cd) = receive cl
  case cd of {
    m : fun ca => close (send (24.3, ca))
}}
```

Fig. 4. Partial server code.

blame, but this blame may depend on the particular label. Thus, each choice coercion is annotated with a blame map β indicating the blame $\beta(l)$ which is raised if a non-existing branch labeled l is addressed. If this blame map is a constant function, then a single blame annotation suffices.

As an example, consider a partial implementation of the server protocol that just serves the current temperature (Fig. 4). If a client sends an r-request, then the choice coercion intercepts this request and triggers blame with label $b.0$. This label abbreviates a second branch in the choice: $\mathtt{r} : 0\#\bot^{b.0}$, a failure coercion that always triggers blame, applied to an arbitrary value. The proxy that implements the "restricting" choice coercion is defined accordingly.

```
case cp of {
  m : fun ca => (df : D, ca1) = receive ca
                ca2 = send (df # num↓^{b.1}, ca1)
                close ca2
  r : 0 # ⊥^{b.0}
}
```

$$
\begin{array}{rcl}
s & ::= & \mathtt{END} \mid \,!\,t.s \mid \,?\,t.s \mid \oplus\langle l : s, \dots\rangle \mid \&\langle l : s, \dots\rangle \\
t & ::= & s \mid * \mid t \otimes t \mid t \multimap t \mid t \to t \mid [s] \\
k & ::= & * \mid c\{s\} \mid n\{s\} \mid \mathtt{request} \mid \mathtt{accept} \mid \mathtt{send} \mid \mathtt{receive} \mid \mathtt{close} \\
e & ::= & x \mid k \mid e \otimes e \mid \mathtt{let}\, x \otimes x = e\, \mathtt{in}\, e \mid \mathtt{rec}\, x(x)e \mid \lambda x.e \mid e\, e \mid \\
 & & \mathtt{select}\, l\, e \mid \mathtt{case}\, e\, \mathtt{of}\, \{l : e, \dots\} \mid \mathtt{fork}\, e\, e \mid \mathtt{pipe}\, s\, e\, e \\
p & ::= & \mathbf{0} \mid e \mid p \parallel p
\end{array}
$$

Fig. 5. Syntax of sessions, types, constants, expressions, and processes.

The server may also implement alternatives that are not required by the protocol, but these extra alternatives require no special handling because the server type is a subtype of the protocol type, in this case.

Dually, the programmer of the client may apply a sender-choice coercion to develop and type check future extensions of a protocol. For example, introducing a new x-request into the client code in Fig. 1 would require adding a line like the following.[3]

```
cl0 = request p
cl = cl0 # !(ι_num). ⊕^{x↦b}_{{x,m,r}} ⟨m : ι_?num.END, r : ι_?num→num.END⟩
cs = send (loc, cl)
...
cx = select x cs
```

An execution that reaches `select x cs` would raise an exception after reducing to `cx # ` $\perp^b$. This execution is impossible with the protocol on `p`.

The final case, where a sender-choice coercion removes an alternative, is dual to the addition of a receiver-choice alternative.

3 Session Calculus with Gradual Types

This section introduces syntax and semantics of the functional calculus with synchronous session types and gradual types. The first subsection explains the fragment without gradual types. It is inspired by Gay and Vasconcelos's calculus for asynchronous functional session types [7]. Our calculus differs from theirs by being synchronous and by describing the semantics with a labeled transition system, which simplifies our proofs. The second subsection introduces coercions and gradual typing; the third subsection defines coercion typing.

3.1 Functional Session Calculus

Fig. 5 defines the syntax of the calculus. A session type s indicates the protocol that can be run on a connection of this type: `END` means that the connection can only be closed, $?\,t.s$ receiving a value of type t and continue according to s ,

[3] The coercion ι_t is the identity coercion at type t. The notation $\mathtt{x} \mapsto b$ denotes a one-element blame map.

$$
\begin{array}{lll}
v & ::= k \mid v \otimes v \mid \mathtt{rec}\, x(x)e \mid \lambda x.e & \overline{\mathtt{END}} = \mathtt{END} \\
\mathcal{E} & ::= \Box \mid \mathcal{E} \otimes e \mid v \otimes \mathcal{E} \mid \mathtt{let}\, x \otimes x = \mathcal{E}\, \mathtt{in}\, e \mid & \overline{!\,t.s} = ?\,t.\bar{s} \\
 & \quad \mathcal{E}\, e \mid v\, \mathcal{E} \mid \mathtt{select}\, l\, \mathcal{E} \mid \mathtt{case}\, \mathcal{E}\, \mathtt{of}\, \{l : e, \dots\} & \overline{?\,t.s} = !\,t.\bar{s} \\
\mathcal{P} & ::= \Box \mid p \parallel \mathcal{P} \mid \mathcal{E} & \overline{\&\langle l : s, \dots\rangle} = \oplus\langle l : \bar{s}, \dots\rangle \\
\alpha & ::= \tau \mid \mathit{req}\, c\{s\} \mid \mathit{acc}\, c\{s\} \mid \mathit{sel}(l, c) \mid \mathit{case}(l, c) \mid & \overline{\oplus\langle l : s, \dots\rangle} = \&\langle l : \bar{s}, \dots\rangle \\
 & \quad !(v, c) \mid ?(v, c) \mid \mathit{close}(c)
\end{array}
$$

Fig. 6. Values, evaluation contexts, process contexts, actions, dual of a session type.

$!\,t.s$ sending a value of type t and continue, $\&\langle l : s, \dots\rangle$ and $\oplus\langle l : s, \dots\rangle$ receive or send a choice with each alternative indexed by a label l and then continuing according to s in that alternative.

A standard type t is either a session type s, a unit type $*$, a linear product $t \otimes t$, a linear function $t \multimap t$, a standard function $t \to t$, or a port $[s]$ which may be used to create connections that use the protocol s.

A constant k is either a unit value $*$, a connection token $c\{s\}$, a port token $n\{s\}$, or a built-in operation involving a connection: `request` and `accept` both take a port argument and produce a connection as described in the reduction rules in Fig. 7. The operations `receive`, `send`, and `close` act on connections in the obvious way. Both connection (channel) and port tokens are special symbols which are adorned with a session type. Connection tokens never show up in source programs and must be handled linearly.

Expressions e are constants, variables, introduction $e \otimes e$ and elimination $\mathtt{let}\, x \otimes x = e\, \mathtt{in}\, e$ of pairs where the latter form is required because the pair type is linear. There are recursive functions $\mathtt{rec}\, x(x)e$, linear functions $\lambda x.e$, function application $e\, e$, as well as $\mathtt{select}\, l\, e$ to send a label indicating a particular alternative in a sender choice, $\mathtt{case}\, e\, \mathtt{of}\, \{l : e, \dots\}$ to perform a choice based on the label received, $\mathtt{fork}\, e_1\, e_2$ to fork e_1 as a new process, and $\mathtt{pipe}\, s\, e_1\, e_2$ to create a channel of type s, fork e_1, and pass the channel ends to e_1 and e_2.

Process expressions p are either the null process $\mathbf{0}$, single expressions e or two processes running in parallel $p \parallel p$. Process expressions are identified up to commutativity and associativity of parallel composition $\parallel$ with the null process $\mathbf{0}$ serving as an identity.

We specify the semantics in small-step operational style using a labeled transition system. Fig. 6 specifies values v as a subset of expressions (constants, pairs of values, and functions), evaluation contexts $\mathcal{E}$ for expressions where $\Box$ is the empty evaluation context (standard strict left-to-right evaluation), and evaluation contexts $\mathcal{P}$ for processes. The latter are understood up to associativity and commutativity of $\parallel$ so that $p \parallel \mathcal{P}$ selects either the left or right subprocess.

Reductions are adorned with actions α, where τ is the silent action (i.e., no interaction), $\mathit{req}\, c\{s\}$ and $\mathit{acc}\, c\{s\}$ signal that a `request` or `accept` operation has been performed on channel $c\{s\}$, similarly, the *sel*, *case*, ?, !, and *close* actions signal that the corresponding operation has been performed on the indicated channel. We usually omit the silent action τ.

$$
\begin{array}{lcl}
\mathcal{P}[\texttt{let}\, x_1 \otimes x_2 = (v_1 \otimes v_2)\, \texttt{in}\, e] & \to & \mathcal{P}[e[x_1, x_2 \mapsto v_1, v_2]] \\
\mathcal{P}[(\texttt{rec}\, x_1(x_2)e_1)\, v] & \to & \mathcal{P}[e_1[x_1, x_2 \mapsto (\texttt{rec}\, x_1(x_2)e_1), v]] \\
\mathcal{P}[(\lambda x.e)\, v] & \to & \mathcal{P}[e[x \mapsto v]]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}[v] & \to & \mathcal{P}[\mathbf{0}] \\
\mathcal{P}[\mathbf{0} \parallel p] & \to & \mathcal{P}[p] \\
\mathcal{P}[\texttt{fork}\, e_1\, e_2] & \to & e_1 \parallel \mathcal{P}[e_2] \\
\mathcal{P}[\texttt{pipe}\, s\, e_1\, e_2] & \to & e_1\, (c\{\bar{s}\}) \parallel \mathcal{P}[e_2\, (c\{s\})] \quad c\{s\} \notin fc(\mathcal{P}, e_1, e_2) \\
\mathcal{P}[\texttt{request}\, n\{s\}] & \xrightarrow{req\ c\{s\}} & \mathcal{P}[c\{s\}] \\
\mathcal{P}[\texttt{accept}\, n\{s\}] & \xrightarrow{acc\ c\{s\}} & \mathcal{P}[c\{\bar{s}\}] \\
\mathcal{P}[\texttt{select}\, l_j\, c\{\oplus\langle l_i : s_i, \ldots\rangle\}] & \xrightarrow{sel(l_j,c)} & \mathcal{P}[c\{s_j\}] \\
\mathcal{P}[\texttt{case}\, c\{\&\langle l_i : \bar{s}_i, \ldots\rangle\}\, \texttt{of}\, \{l_i : e_i, \ldots\}] & \xrightarrow{case(l_j,c)} & \mathcal{P}[e_j\, c\{\bar{s}_j\}] \\
\mathcal{P}[\texttt{send}\, (v \otimes c\{!\, t.s\})] & \xrightarrow{!(v,c)} & \mathcal{P}[c\{s\}] \\
\mathcal{P}[\texttt{receive}\, c\{?\, t.\bar{s}\}] & \xrightarrow{?(v,c)} & \mathcal{P}[v \otimes c\{\bar{s}\}] \\
\mathcal{P}[\texttt{close}\, c\{\texttt{END}\}] & \xrightarrow{close(c)} & \mathcal{P}[*]
\end{array}
$$

$$\textsc{RClose}\quad \frac{p_1 \xrightarrow{close(c)} p_1' \qquad p_2 \xrightarrow{close(c)} p_2'}{\mathcal{P}[p_1 \parallel p_2] \to \mathcal{P}[p_1' \parallel p_2']}$$

$$\textsc{RSingle}\quad \frac{p_1 \xrightarrow{!(v,c)} p_1' \qquad p_2 \xrightarrow{?(v,c)} p_2'}{\mathcal{P}[p_1 \parallel p_2] \to \mathcal{P}[p_1' \parallel p_2']}$$

$$\textsc{RChoice}\quad \frac{p_1 \xrightarrow{sel(l,c)} p_1' \qquad p_2 \xrightarrow{case(l,c)} p_2'}{\mathcal{P}[p_1 \parallel p_2] \to \mathcal{P}[p_1' \parallel p_2']}$$

$$\textsc{RConnect}\quad \frac{p_1 \xrightarrow{req\ c\{s\}} p_1' \qquad p_2 \xrightarrow{acc\ c\{s\}} p_2' \qquad c\{s\} \notin fc(\mathcal{P}[p_1 \parallel p_2])}{\mathcal{P}[p_1 \parallel p_2] \to \mathcal{P}[p_1' \parallel p_2']}$$

Fig. 7. Evaluation rules.

Fig. 7 specifies the top-level reduction relation $\to$ which relies on an auxiliary labeled reduction relation where the label specifies a communication action α. The context rules RClose, RSingle, RChoice, and RConnect guarantee that these labeled reductions always pair up one producer action and one consumer action. The RConnect rule guarantees that the newly created connection token is fresh, using $fc(p)$ for the set of connection tokens in process p. The labeled reduction rules are *not* intended to run at the top-level. For each of them, there is a context rule that matches it up with a corresponding labeled reduction (labeled with the complementary action on the same channel) in another process.

The reduction rules for let, rec, and lambda are standard. A process that has been reduced to a value turns into the null process. The null process in parallel to some p is eliminated. The $\texttt{fork}\, e_1\, e_2$ expression starts e_1 as a new process. The $\texttt{pipe}\, s\, e_1\, e_2$ expression creates a fresh channel $c\{s\}$, passes the server end to e_1 and the client end to e_2, and then starts the server $e_1(c\{\bar{s}\})$ as a new process. The labeled reductions for `request` and `accept` work together with the RConnect rule to create a new connection. The RConnect rule makes sure that the same connection token is used with the same session type in both contracta.

Additionally, the `accept` rule emits the server end of the channel where the session type is inverted (dualized). This inversion is indicated by the function $\bar{s}$ (see Figure 7). It exchanges the read and write operations in a protocol to generate the server view from the client view.

The reductions for `select` and `case` work together with RChoice to select an alternative in a session choice. They update the session type of the connection token accordingly. The reductions for `send` and `receive` work together with RSingle in the same way. The `send` operation takes a linear pair and the `receive` operation returns one because these pairs contain a connection token, which is a linear value.

The reduction for `close` consumes the connection token at returns the unit value. The RClose rule guarantees that exactly two processes perform a close step on both ends of the same connection.

Figure 8 contains the definition of the type system. It first defines symbol environments Σ that associate channel names with session types and typing environments Γ that map identifiers to types. The use of Γ_x in the definition indicates that x does not occur in Γ_x. In contrast, a symbol environment may contain multiple identical associations for the same channel. The predicate $unr(t)$ specifies unrestricted types, which need not be treated linearly, along with its extension $unr(\Gamma)$ to typing environments. The splitting operator + for environments is defined as usual to avoid the duplication and weakening of linearly typed values. We also use splitting for symbol environments.

The calculus contains type and session type indexed families of constants for the operations request, accept, send, receive, and close. The function $hd(s)$ describes the next possible communication on a channel of type s. It may be a set of labels, with the empty set denoting that the channel may only be closed, or the type of the next value that may be transmitted on the channel.

The typing rules are standard for a lambda calculus with linear types. The rule for a channel makes sure its type is specified by the symbol environment. The rules for the `select` and `case` constructs are standard for the session choice operators in a calculus with session types. Because connections are represented by type-carrying connection tokens, the typing judgment for processes does not require a typing environment, but the symbol environment is required to ensure consistent use of channels.

3.2 Gradual Typing

Gradual typing adds a type D "dynamic" to the type language, introduces a type cast operation of the form $e\#\gamma$, and a blame signal $\Uparrow^b$ to the expression language. A cast checks the actual type of a value of type dynamic at run time. A value of dynamic type has the form $D_{tc}(v)$, which is a pair of a type constructor and a ground type value v of type $tc(\overrightarrow{D})$. The new syntactic form γ is a coercion defined along with the other syntactic extensions by the grammar in Figure 9. Type constructors (except session types) are ranged over by tc, each with an associated arity. Ground types g are applications of a type constructor to all dynamic arguments.

Symbol environments, variable environments, unrestricted types

$$\Sigma ::= \cdot \mid \Sigma, c : s \qquad \Gamma ::= \cdot \mid \Gamma_x, x : t \qquad unr(*) \qquad unr(t \to t) \qquad unr([s])$$

Unrestricted environments

$$unr(\cdot) \qquad \frac{unr(\Gamma) \qquad unr(t)}{unr(\Gamma, x : t)}$$

Environment splitting

$$\cdot + \cdot = \cdot \qquad \frac{unr(t) \qquad \Gamma_1 + \Gamma_2 = \Gamma}{x : t, \Gamma_1 + x : t, \Gamma_2 = x : t, \Gamma}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma}{x : t, \Gamma_1 + \Gamma_2 = x : t, \Gamma} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1 + x : t, \Gamma_2 = x : t, \Gamma}$$

Typing of constants

$$\texttt{request} : [s] \to s \qquad \texttt{accept} : [s] \to \bar{s}$$

$$\texttt{send} : t \otimes (!\, t.s) \to s \qquad \texttt{receive} : (?\, t.s) \to t \otimes s \qquad \texttt{close} : \texttt{END} \to *$$

Head of session type

$$hd(\texttt{END}) = \emptyset \qquad hd(?\, t.s) = hd(!\, t.s) = t$$

$$hd(\&\langle l : s, \dots \rangle) = hd(\oplus\langle l : s, \dots \rangle) = \{l, \dots\}$$

Typing rules for expressions

$$\frac{unr(\Gamma)}{\cdot, \Gamma + x : t \vdash x : t} \qquad \frac{unr(\Gamma)}{\cdot, \Gamma \vdash * : *} \qquad \frac{unr(\Gamma)}{c : s, \Gamma \vdash c\{s\} : s} \qquad \frac{unr(\Gamma)}{\cdot, \Gamma \vdash n\{s\} : [s]}$$

$$\frac{\Sigma_1, \Gamma_1 \vdash e_1 : t_1 \qquad \Sigma_2, \Gamma_2 \vdash e_2 : t_2}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash e_1 \otimes e_2 : t_1 \otimes t_2}$$

$$\frac{\Sigma_1, \Gamma_1 \vdash e_1 : t_1 \otimes t_2 \qquad \Sigma_2, \Gamma_2 + x_1 : t_1, x_2 : t_2 \vdash e : t}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash \texttt{let}\, x_1 \otimes x_2 = e_1\, \texttt{in}\, e : t}$$

$$\frac{unr(\Gamma) \qquad \cdot, \Gamma + x_1 : t_2 \to t, x_2 : t_2 \vdash e : t}{\cdot, \Gamma \vdash \texttt{rec}\, x_1(x_2)e : t_2 \to t} \qquad \frac{\Sigma, \Gamma + x : t_1 \vdash e : t_2}{\Sigma, \Gamma \vdash \lambda x.e : t_1 \multimap t_2}$$

$$\frac{\Sigma_1, \Gamma_1 \vdash e_1 : t_2 \to t \qquad \Sigma_2, \Gamma_2 \vdash e_2 : t_2}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash e_1\, e_2 : t} \qquad \frac{\Sigma_1, \Gamma_1 \vdash e_1 : t_2 \multimap t \qquad \Sigma_2, \Gamma_2 \vdash e_2 : t_2}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash e_1\, e_2 : t}$$

$$\frac{\Sigma, \Gamma \vdash e : \oplus\langle l : s, \dots \rangle}{\Sigma, \Gamma \vdash \texttt{select}\, l\, e : s} \qquad \frac{\Sigma_1, \Gamma_1 \vdash e : \&\langle l_i : s_i, \dots \rangle \qquad \Sigma_2, \Gamma_2 \vdash e_i : s_i \multimap t}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash \texttt{case}\, e\, \texttt{of}\, \{l_i : e_i, \dots\} : t}$$

$$\frac{\Sigma_1, \Gamma_1 \vdash e_1 : * \qquad \Sigma_2, \Gamma_2 \vdash e_2 : t_2}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash \texttt{fork}\, e_1\, e_2 : t_2} \qquad \frac{\Sigma_1, \Gamma_1 \vdash e_1 : \bar{s} \multimap * \qquad \Sigma_2, \Gamma_2 \vdash e_2 : s \multimap t_2}{\Sigma_1 + \Sigma_2, \Gamma_1 + \Gamma_2 \vdash \texttt{pipe}\, s\, e_1\, e_2 : t_2}$$

Typing rules for processes

$$\cdot \vdash \mathbf{0} \qquad \frac{unr(t) \qquad \Sigma, \cdot \vdash e : t}{\Sigma \vdash e} \qquad \frac{\Sigma_1 \vdash p_1 \qquad \Sigma_2 \vdash p_2}{\Sigma_1 + \Sigma_2 \vdash p_1 \parallel p_2}$$

Fig. 8. Typing rules and auxiliary definitions.

$$
\begin{array}{ll}
t ::= \cdots \mid D & tc ::= * \mid \otimes \mid \ \multimap \ \mid \ \to \ \mid [s] \\
e ::= \cdots \mid e\#\gamma \mid \Uparrow^b & g \ ::= * \mid D \otimes D \mid D \multimap D \mid D \to D \mid [s] \\
\mathcal{E} ::= \cdots \mid \mathcal{E}\#\gamma & \gamma \ ::= \sigma \mid \iota_t \mid g\uparrow \mid g\downarrow^b \mid \gamma\,;\gamma \mid \gamma \otimes \gamma \mid \gamma \multimap \gamma \mid \gamma \to \gamma \mid \bot^b \\
v ::= \cdots \mid D_{tc}(v) & \sigma \ ::= \mathtt{END} \mid \,!\,\gamma.\sigma \mid \,?\,\gamma.\sigma \mid \oplus^{\beta}_{\mathfrak{L}}\langle l : \sigma, \ldots\rangle \mid \&^{\beta}_{\mathfrak{L}}\langle l : \sigma, \ldots\rangle
\end{array}
$$

Fig. 9. Syntax extensions for gradual typing: type constructors, ground types, coercions, and session coercions.

A coercion term γ is either a session coercion σ, an identity coercion ι_t indexed by a type, an injection of a ground type into type D, a projection from D into a ground type, diagrammatic (left-to-right) composition of coercions $\gamma;\gamma$, functorial coercions that apply coercions under the type constructors for pair, linear function, and function, or a coercion $\bot^b$ that always fails. It is indexed with the blame label b that is raised when the coercion is executed. The projection is also indexed with a blame label that is raised if the underlying dynamic value does not match the expected ground type.

Session type coercions σ mimic the syntax of session types, but with coercions in place of types. As with subtyping, sending type positions in session types and coercions are covariant whereas receiving positions are contravariant. The choice coercions carry a blame map b and their range index set $\mathfrak{L}$ as annotations. They facilitate the addition and removal of alternative choices.

The send choice coercion $\oplus^{\beta}_{\mathfrak{M}}\langle l : \sigma \mid l \in \mathfrak{L}\rangle$ applies to a channel with alternatives at least $\mathfrak{L}$ and it provides a channel with alternatives $\mathfrak{M}$. Only the coercions for the $\mathfrak{L}$ continuations need to be provided, the others generate blame according to the blame map β. The domain of β is $\mathfrak{M} \setminus \mathfrak{L}$. Dually, the receive choice coercion $\&^{\beta}_{\mathfrak{M}}\langle l : \sigma \mid l \in \mathfrak{L}\rangle$ applies to a connection with at most $\mathfrak{M}$ alternatives and it provides a channel with $\mathfrak{L}$ alternatives. Only labels in $\mathfrak{L}$ are forwarded, the others generate blame according to β. The domain of β is again $\mathfrak{M} \setminus \mathfrak{L}$.

3.3 Coercion Typing

Coercions come with their own type system which derives judgments of the form $\gamma : t \Rightarrow t$. Figure 10 contains the typing rules for coercions along with expression typing rules for coercion application and the blame expression. It remains to define the semantics of the coercions in Figures 11 and 12.

Applying a composed coercion gets reduced to a nested application of the components. We split a composed coercion in this way to simplify the creation of proxies: they only have to deal with simple, non-composed coercions at the top-level. The identity coercion leaves its argument unchanged. The injection of a ground type into D adds the type tag to the value. The projection from dynamic checks the type constructor and either yields the value if the type constructors coincide or raises a blame exception, otherwise. Applying the failure coercion directly raises blame. A functorial coercion expands its argument and applies the constituent coercions to the constituents of the argument. A session coercion

Typing rules for coercion terms

$$\iota_t : t \Rightarrow t \qquad g{\uparrow}\colon g \Rightarrow D \qquad g{\downarrow^b}\colon D \Rightarrow g \qquad \frac{\gamma_1 : t_0 \Rightarrow t_1 \qquad \gamma_2 : t_1 \Rightarrow t_2}{\gamma_1 \,;\, \gamma_2 : t_0 \Rightarrow t_2}$$

$$\frac{\gamma : t_1' \Rightarrow t_1 \qquad \gamma' : t_2 \Rightarrow t_2'}{\gamma \to \gamma' : t_1 \to t_2 \Rightarrow t_1' \to t_2'} \qquad \frac{\gamma : t_1' \Rightarrow t_1 \qquad \gamma' : t_2 \Rightarrow t_2'}{\gamma \multimap \gamma' : t_1 \multimap t_2 \Rightarrow t_1' \to t_2'}$$

$$\frac{\gamma_i : t_i \Rightarrow t_i'}{\gamma_1 \otimes \gamma_2 : t_1 \otimes t_2 \Rightarrow t_1' \otimes t_2'} \qquad \bot^b : t \Rightarrow t'$$

Typing rules for session coercions

$$\iota_s : s \Rightarrow s \qquad \frac{\gamma : t \Rightarrow t' \qquad \sigma : s \Rightarrow s'}{!\,\gamma.\sigma : !\,t.s \Rightarrow !\,t'.s'} \qquad \frac{\gamma : t' \Rightarrow t \qquad \sigma : s \Rightarrow s'}{?\,\gamma.\sigma : ?\,t.s \Rightarrow ?\,t'.s'}$$

$$\frac{(\forall l \in \mathfrak{L})\sigma_l : s_l \Rightarrow s_l' \qquad dom(\beta) = \mathfrak{M} \setminus \mathfrak{L}}{\oplus^{\beta}_{\mathfrak{M}}\langle l : \sigma_l \mid l \in \mathfrak{L}\rangle : \oplus\langle l : s_l \mid l \in \mathfrak{L}\rangle \Rightarrow \oplus\langle l : s_l' \mid l \in \mathfrak{M}\rangle}$$

$$\frac{(\forall l \in \mathfrak{L})\sigma_l : s_l \Rightarrow s_l' \qquad dom(\beta) = \mathfrak{M} \setminus \mathfrak{L}}{\&^{\beta}_{\mathfrak{M}}\langle l : \sigma_l \mid l \in \mathfrak{L}\rangle : \&\langle l : s_l \mid l \in \mathfrak{M}\rangle \Rightarrow \&\langle l : s_l' \mid l \in \mathfrak{L}\rangle}$$

Expression typing rules

$$\frac{\Sigma, \Gamma \vdash e : t \qquad \gamma : t \Rightarrow t'}{\Sigma, \Gamma \vdash e\#\gamma : t'} \qquad \frac{}{\Sigma, \Gamma \vdash \Uparrow^b : t}$$

Fig. 10. Coercion typing.

creates a new connection of the target type of the coercion and it forks a new process running the expression generated by the meta-function $\mathit{Proxy}(\mathbf{x}, \sigma, \mathbf{y})$ from the coercion σ. This process serves as a proxy between the source-typed connection $\mathbf{x}$ and the target-typed connection $\mathbf{y}$.

The generation of the proxy expression is straightforward (Fig. 12) except for two special twists. One point is the generation of code from an identity coercion ι_s. It continues by mapping the type into the corresponding eta-expanded identity coercion $\hat{s}\colon \widehat{!\,t.s} = !\,\iota_t.\hat{s}$, $\widehat{\oplus\langle l : s_l \mid l \in \mathfrak{L}\rangle} = \oplus^{\emptyset}_{\mathfrak{L}}\langle l : \hat{s}_l \mid l \in \mathfrak{L}\rangle$, and analogously for the remaining session type constructors.

The other point is the consumption of the two linearly typed channels in the choice coercions that discard alternatives. In these cases, the proxy code bundles the channels into a linear pair and applies a failure coercion to the pair.

4 Type Preservation and Progress Properties

This section considers the interplay between the operational semantics and the type system. We show that reduction of expressions and processes preserves their types. We further characterize the conditions under which typed expressions and processes make progress. We only consider the extended calculus with gradual types and start by establishing preservation and progress for expressions.

$$
\begin{array}{llll}
v\#(\gamma\,;\gamma') \to (v\#\gamma)\#\gamma' & & D_{tc}(v)\#tc{\downarrow^b} \to v & \\
v\#\iota_t \to v & & D_{tc}(v)\#tc'{\downarrow^b} \to \Uparrow^b & tc \neq tc' \\
v\#tc{\uparrow} \to D_{tc}(v) & & v\#\bot^b \to \Uparrow^b &
\end{array}
$$

$$
\begin{array}{ll}
v\#(\gamma \otimes \gamma') & \to \mathtt{let}\, x_1 \otimes x_2 = v\, \mathtt{in}\, (x_1\#\gamma) \otimes (x_2\#\gamma') \\
v\#(\gamma \to \gamma') & \to \mathtt{rec}\, f(y)(v\,(y\#\gamma))\#\gamma' \\
v\#(\gamma \multimap \gamma') & \to \lambda y.(v\,(y\#\gamma))\#\gamma' \\
v\#\sigma & \to \mathtt{pipe}\, s'\, (\lambda y.\mathit{Proxy}(v,\sigma,y))\, (\lambda y.y) \\
 & \quad \mathit{if}\ \sigma : s \Rightarrow s'
\end{array}
$$

Fig. 11. Eager cast reduction (in context $\mathcal{P}$ or $\mathcal{E}$).

$$
\begin{array}{ll}
\mathit{Proxy}(\mathbf{x}, \iota_s, \mathbf{y}) & = \mathit{Proxy}(\mathbf{x}, \hat{s}, \mathbf{y}) \\
\mathit{Proxy}(\mathbf{x}, \mathtt{END}, \mathbf{y}) & = \mathtt{close}\,\mathbf{x}; \mathtt{close}\,\mathbf{y} \\
\mathit{Proxy}(\mathbf{x}, ?\,\gamma.\sigma, \mathbf{y}) & = \mathtt{let}\, x_1 \otimes x_2 = \mathtt{receive}\,\mathbf{x}\,\mathtt{in}\,\mathtt{let}\, y = \mathtt{send}\,((x_1\#\gamma) \otimes \mathbf{y})\,\mathtt{in} \\
 & \quad \mathit{Proxy}(x_2, \sigma, y) \\
\mathit{Proxy}(\mathbf{x}, !\,\gamma.\sigma, \mathbf{y}) & = \mathit{Proxy}(\mathbf{y}, ?\,\gamma.\bar{\sigma}, \mathbf{x}) \\
\mathit{Proxy}(\mathbf{x}, \&^{\beta}_{\mathfrak{M}}\langle l : \sigma_l \mid l \in \mathfrak{L}\rangle, \mathbf{y}) & \\
 \quad = \mathtt{case}\,\mathbf{x}\,\mathtt{of}\,\{\, l : \lambda x.\mathtt{let}\, y = \mathtt{select}\, l\,\mathbf{y}\,\mathtt{in}\,\mathit{Proxy}(x, \sigma_l, y) \mid l \in \mathfrak{L}, & \\
 \qquad l : \lambda x.(x \otimes \mathbf{y})\#\bot^{\beta(l)} \mid l \in \mathfrak{M} \setminus \mathfrak{L}\,\} & \\
\mathit{Proxy}(\mathbf{x}, \oplus^{\beta}_{\mathfrak{M}}\langle l : \sigma_l \mid l \in \mathfrak{L}\rangle, \mathbf{y}) & = \mathit{Proxy}(\mathbf{y}, \&^{\beta}_{\mathfrak{M}}\langle l : \bar{\sigma}_l \mid l \in \mathfrak{L}\rangle, \mathbf{x})
\end{array}
$$

Fig. 12. Coercion proxies.

Lemma 1 (Preservation I). *Suppose that $\Sigma, \cdot \vdash e : t$ and that $e \xrightarrow{\alpha} e'$. Then there exists some Σ' such that $\Sigma', \cdot \vdash e' : t$.*

Lemma 2 (Progress I). *If $\Sigma, \cdot \vdash e : t$, then one of the following holds: 1. e is a value; 2. $e = \Uparrow^b$ raises blame; 3. $\exists e'$ such that $e \to e'$; 4. $e = \mathcal{E}[\mathtt{fork}\, e_1\, e_2]$; 5. $e = \mathcal{E}[\mathtt{pipe}\, s\, e_1\, e_2]$; 6. $e = \mathcal{E}[\mathtt{request}\, n\{s\}]$; 7. $e = \mathcal{E}[\mathtt{accept}\, n\{s\}]$; 8. $e = \mathcal{E}[\mathtt{select}\, l_j\, c\{\oplus\langle l_i : s_i, \ldots\rangle\}]$; 9. $e = \mathcal{E}[\mathtt{case}\, c\{\&\langle l_i : \bar{s}_i, \ldots\rangle\}\,\mathtt{of}\,\{l_i : e_i, \ldots\}]$; 10. $e = \mathcal{E}[\mathtt{send}\,(v \otimes c\{!\,t.s\})]$; 11. $e = \mathcal{E}[\mathtt{receive}\, c\{?\,t.\bar{s}\}]$; 12. $e = \mathcal{E}[\mathtt{close}\, c\{\mathtt{END}\}]$.*

Preservation for expressions extends to processes.

Lemma 3 (Preservation II). *Suppose that $\Sigma \vdash p$ and $p \to p'$. Then there exists some Σ' such that $\Sigma' \vdash p'$.*

Lemma 4 (Progress II). *Suppose that $\Sigma \vdash p$. Then one of the following holds: 1. $p = \mathbf{0}$; 2. $p = \Uparrow^b$; 3. $\exists p'$ such that $p \to p'$; 4. $p = p_1 \parallel p_2 \parallel p_3$ where*
- *$p_1 = \mathcal{E}[\mathtt{request}\, n_1\{s\}] \parallel \ldots \parallel \mathcal{E}[\mathtt{request}\, n_{k_2}\{s\}]$,*
- *$p_2 = \mathcal{E}[\mathtt{accept}\, n_{k_2+1}\{s\}] \parallel \ldots \parallel \mathcal{E}[\mathtt{accept}\, n_{k_2+k_3}\{s\}]$*
- *$\{n_1, \ldots, n_{k_1}\} \cap \{n_{k_1+1}, \ldots, n_{k_1+k_2}\} = \emptyset$,*
- *$p_3 = \mathcal{E}[N_1[c_1\{s\}]] \parallel \ldots \parallel \mathcal{E}[N_{k_4}[c_{k_4}\{s\}]]$ where all c_j are distinct and*

$N ::= \mathtt{select}\, l_j\, \Box \mid \mathtt{case}\, \Box\, \mathtt{of}\, \{l_j : e_j, \ldots\} \mid \mathtt{send}\,(v \otimes \Box) \mid \mathtt{receive}\, \Box \mid \mathtt{close}\, \Box$

It is understood that $p_1 = \mathbf{0}$ if $k_1 = 0$ and analogously for p_2, p_3, and p_4.

All proofs for the preservation results (Lemmas 1 and 3) proceed by induction on the reduction relation. The progress results (Lemma 2 and 4) are by induction

on the typing derivation. The proof of the last lemma exploits associativity and commutativity of the $\|$ operator as well as the neutrality of $\mathbf{0}$.

Typing does not guarantee progress. The creation of a connection via request and accept guarantees that the two ends of a channel start of in different processes. However, this interaction may never happen (p_1 and p_2). Furthermore, due to the presence of higher-order channels, a process may receive the other end of a channel that it already works with. Any action on one of the ends leads to deadlock according to p_5 in Lemma 4.

5 Related Work

The introduction gives an overview of a small fraction of the work on session types. This work is the first to explore the connection of session types with gradual types. A further novelty is the exploration of gradual typing in the presence of choice types.

Gradual types are also briefly explored in the introduction. Here, we offer some additional perspective on the use of coercions in such a calculus. Henglein [9] introduced the coercion calculus to investigate the foundations of dynamic typing and in particular the optimization possibilities from manipulating type tagging and untagging operations. Early work on gradual types [16] did not use coercions, however, the need for space efficient representations [10] required a normalization procedure for compositions of casts. This effort culminated in the discovery of threesomes [17], roughly casts with an intermediate type, which are a compressed representation of arbitrary sequences of simple casts. The same paper also shows that threesomes are intertranslatable with coercions.

6 Conclusions and Future Work

We define the first calculus with session types and gradual types. The calculus provides useful insights for interlanguage programs, where one end of a protocol is run by a statically typed program and the other end by a dynamically typed one. It introduces a novel, liberal notion of cast between session types. We establish its basic metatheory and explore an eager implementation casts with proxies. In future work, we want to consider a different, lazy implementation of casts with commuting conversions as well as further extensions to the calculus.

References

1. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236, Paris, France, Aug. 2010. Springer.
2. M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and session types. *Information and Computation*, 207(5):595–641, 2009.
3. T. Disney and C. Flanagan. Gradual information flow typing. In *STOP*, 2011.

4. L. Fennell and P. Thiemann. The blame theorem for a linear lambda calculus with type dynamic. In H.-W. Loidl and R. Peña, editors, *TFP 2012*, volume 7829 of *LNCS*, pages 37–52, St Andrews, UK, June 2012. Springer.
5. L. Fennell and P. Thiemann. Gradual security typing with references. In V. Cortier and A. Datta, editors, *CSF*, pages 224–239, New Orleans, LA, USA, 2013. IEEE.
6. S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *Proc. 1999 ESOP*, volume 1576 of *LNCS*, pages 74–90, Amsterdam, The Netherlands, Apr. 1999. Springer.
7. S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.
8. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In POPL 2010 [15], pages 299–312.
9. F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.
10. D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, 2007.
11. K. Honda. Types for dyadic interaction. In E. Best, editor, *Proceedings of 4th International Conference on Concurrency Theory*, number 715 in LNCS, pages 509–523, Aug. 1993.
12. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT 2011*, volume 6536 of *LNCS*, pages 55–75, Bhubaneshwar, India, 2011. Springer.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In P. Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 273–284, San Francisco, CA, USA, Jan. 2008. ACM Press.
14. R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda. Practical interruptible conversations - distributed dynamic verification with session types and python. In *RV*, volume 8174 of *LNCS*, pages 130–148, Rennes, France, Sept. 2013. Springer.
15. *Proc. 37th ACM Symp. POPL*, Madrid, Spain, Jan. 2010. ACM Press.
16. J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.
17. J. G. Siek and P. Wadler. Threesomes, with and without blame. In POPL 2010 [15], pages 365–376.
18. J. A. Tov and R. Pucella. Stateful contracts for affine types. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *LNCS*, pages 550–569. Springer, 2010.
19. V. T. Vasconcelos, A. Ravara, and S. J. Gay. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1-2):64–87, 2006.
20. P. Wadler. Propositions as sessions. In R. B. Findler, editor, *ICFP'12*, pages 273–286, Copenhagen, Denmark, Sept. 2012. ACM.
21. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer.
22. R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP*, volume 6813 of *LNCS*, pages 459–483, Lancaster, UK, 2011. Springer.