Context-Free Session Type Inference

Context-Free Session Type Inference

LUCA PADOVANI, Dipartimento di Informatica, Università di Torino, ITALY

Some interesting communication protocols can be precisely described only by context-free session types, an extension of conventional session types supporting a general form of sequential composition. The complex metatheory of context-free session types, however, hinders the definition of corresponding checking and inference algorithms. In this work we study a new syntax-directed type system for context-free session types that is easy to embed into a host programming language. We also detail two `OCaml` embeddings which allow us to piggyback on `OCaml`'s type system to check and infer context-free session types.

CCS Concepts: • **Theory of computation → Type structures**; *Distributed computing models*; • **Software and its engineering → Automated static analysis**; **Concurrent programming structures**; *Functional languages*;

Additional Key Words and Phrases: context-free session types, existential types, `OCaml`, session type inference

ACM Reference Format:
Luca Padovani. 2017. Context-Free Session Type Inference. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2017), 36 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Session types [11, 12, 14] are an established formalism for the enforcement of communication protocols through static analysis. Recently, Thiemann and Vasconcelos [31] have proposed *context-free session types* to enhance the expressiveness of conventional session types. The distinguishing feature of context-free session types is a general form of sequential composition $T;S$ to compose arbitrary sub-protocols T and S instead of restricting T to a single input/output action as is commonly the case. Protocols that benefit from such enhancement include the serialization of tree-like data structures and XML documents [31], interactions with non-uniform objects such as stacks and reentrant locks [6, 24], and recursive protocols for trust management [30]. Thiemann and Vasconcelos [31] study the metatheory of context-free session types, leaving the definition of a type checking algorithm for future work. In this paper we point out additional issues that afflict context-free session type checking and inference and we describe a practical solution to their implementation in `OCaml` and other general-purpose programming languages.

We illustrate a concrete case where context-free session types are useful by means of the `OCaml` code shown in Figure 1, which uses the `FuSe` library for binary sessions [26]. The `stack` function models a stack as a non-uniform object [24, 28] offering different interfaces through a session endpoint `u` depending on its internal state. An empty stack (lines 2–6) accepts either a `Push` or an `End` message. In the first case, the stack `receive`s the element `x` to be pushed and moves into the non-empty state with the recursive application `some x u`. In the second case, it just returns the endpoint `u`. Note that the element pushed into the stack is stored in the frame of `some` rather than

Author's address: Luca Padovani, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, Torino, 10149, ITALY, luca.padovani@unito.it.

0164-0925/2017/1-ART1 $15.00
https://doi.org/0000001.0000001

```
1  let stack =
2    let rec none u =
3      match branch u with
4      | `Push u → let x, u = receive u
5                  in none (some x u)
6      | `End u → u
7    and some y u =
8      match branch u with
9      | `Push u → let x, u = receive u
10                 in some y (some x u)
11     | `Pop u → send y u
12   in none
```

Fig. 1. Modeling of a non-uniform object accessed through a session.

in a data structure. A non-empty stack (lines 7–11) with y on top accepts either a Push message, as in the empty case, or a Pop message, in which case it sends y back to the client. In both cases, when an application some x u terminates, meaning that x has been popped, the stack returns to its previous state, whatever it was (lines 5 and 10). Initially, the stack is empty (line 12). Note that, according to established conventions [10], all session primitives including send return the endpoint u possibly paired with the received message (receive) or injected through a tag that represents an operation (branch). This allows the endpoint u to be associated with different session types in different parts of the program where it is being used in accordance with the communication protocol. Using the FuSe implementation of binary sessions [26], OCaml infers for stack the type $S_{\mathtt{reg}} \to \beta$ where $S_{\mathtt{reg}}$ is the (equi-recursive) session type that satisfies the equation

$$S_{\mathtt{reg}} = \&[\mathsf{Push} : ?\alpha ; S_{\mathtt{reg}}] \tag{1}$$

according to which the client can only push elements of type α. In particular, this type forbids the client to ever send End or Pop messages to the stack, even though the code in Figure 1 does allow these operations at precise times during the lifetime of a stack. To understand the reason why the End and Pop operations do not occur in $S_{\mathtt{reg}}$, we have to consider that conventional session types can only describe protocols whose set of (finite) traces is regular, whereas the set of (finite) traces that describe legal interactions with stack is isomorphic to the language of balanced parentheses, a typical example of a context-free language that is not regular. The session type $S_{\mathtt{reg}}$ above corresponds to the best ω-regular and safe approximation of this context-free language that OCaml manages to infer from the code of stack. When OCaml figures that the session type cannot precisely track whether the stack is empty or not, and therefore whether an End or Pop operation is allowed, it computes the "intersection" of the interfaces of these two states, which results in $S_{\mathtt{reg}}$ along with warnings informing that lines 6 and 11 are dead code.

Guided by analogous considerations, Thiemann and Vasconcelos [31] propose *context-free session types* as a more expressive protocol description language. The key idea is to enforce the order of interactions in a protocol using a general form of sequential composition _;_ instead of the usual prefix operator. For example, the context-free session types $S_{\mathtt{none}}$ and $S_{\mathtt{some}}$ that satisfy the equations

$$\begin{array}{rcl} S_{\mathtt{none}} & = & \&[\mathsf{Push} : ?\alpha ; S_{\mathtt{some}} ; S_{\mathtt{none}}, \ \mathsf{End} : \mathbf{1}] \\ S_{\mathtt{some}} & = & \&[\mathsf{Push} : ?\alpha ; S_{\mathtt{some}} ; S_{\mathtt{some}}, \ \mathsf{Pop} : !\alpha] \end{array} \tag{2}$$

provide accurate descriptions of the legal interactions with `stack`: all finite, maximal traces described by S_{none} have each Push eventually followed by a matching Pop. The "empty" protocol $\mathbf{1}$ marks the end of a legal interaction. Using Thiemann and Vasconcelos' type system, it is then possible to work out a typing derivation showing that `stack` has type $S_{\mathsf{none}}\,;A \to A$, where A is a session type variable that can be instantiated with any session type.

In the present work we propose a practical realization of Thiemann and Vasconcelos' type system that ultimately allows us to *infer* a type as precise as $S_{\mathsf{none}}\,;A \to A$ from the code of a function very similar to (but not exactly the same as) `stack`. There are two key features of Thiemann and Vasconcelos' type system that make it challenging to be used as the basis for a type inference algorithm: (1) a structural rule that rearranges session types according to the monoidal and distributive laws of sequential composition and (2) the need to support polymorphic recursion which ultimately arises as a consequence of (1). Type inference in the presence of polymorphic recursion is known to be undecidable in general [16], a problem which often requires programmers to explicitly annotate polymorphic-recursive functions with their type. In addition, the liberal handling of sequential compositions means that functions like `stack` admit very different types (such as $S_{\mathtt{reg}} \to \beta$ and $S_{\mathsf{none}}\,;A \to A$) which do not appear to be instances of a unique, more general type scheme. It is therefore unclear which notion of principal type should guide a type inference algorithm. There is a third observation that must be kept in mind: even though Thiemann and Vasconcelos [31] show that context-free session type equivalence is decidable, subtyping is not (we give a proof of this fact in the present paper). Therefore, every direct realization of the type system in [31] would have to disallow or at least restrict subtyping, a feature that has been extensively studied in the session type literature and happens to be practically relevant [9, 10, 23].

Overall, these observations lead us to reconsider the way sequential compositions are handled by the type system. Our proposal is to eliminate sequential compositions through an explicit, higher-order combinator `@>` called *resumption* that has the following signature

$$\texttt{@>} : (T \to \mathbf{1}) \to T\,;S \to S \tag{3}$$

and that is operationally akin to functional application. Suppose $f : T \to \mathbf{1}$ is a function that, applied to a session endpoint of type T, carries out the communication over the endpoint and returns the depleted endpoint, of type $\mathbf{1}$. Using `@>` we can supply to f an endpoint u of type $T\,;S$ knowing that f will take care of the prefix T of $T\,;S$ and leaving us with an endpoint of type S. In other words, `@>` allows us to modularize the enforcement of a sequential protocol $T\,;S$ by partitioning the program into a part – the function f – that carries out the prefix T of the protocol and another part – the evaluation context in which f `@>` u occurs – that carries out the continuation S.

From this informal description of the semantics of `@>` it is possible to spot a potential flaw of our approach. The type $T \to \mathbf{1}$ describes a function that takes an endpoint of type T and returns an endpoint of type $\mathbf{1}$, but does not guarantee that the returned endpoint is *the same endpoint* supplied to the function. Only in this case we are guaranteed that the sub-protocol T has been completed and that the endpoint can be safely resumed. In order to reason about such guarantee, we need a type-level mechanism to capture the *identity* of endpoints. Similar requirements have already arisen in different contexts, to identify regions [2, 36] and to associate resources with capabilities [1, 32, 34]. Reframing the techniques used in these works to our setting, the idea is to refine endpoint types to a form $[T]_\varrho$ where ϱ is a variable that represents the abstract identity of the endpoint at the type level. The refined signature of `@>` therefore becomes

$$\texttt{@>} : ([T]_\varrho \to [\mathbf{1}]_\varrho) \to [T\,;S]_\varrho \to [S]_\varrho \tag{4}$$

where the fact that the *same* ϱ decorates both $[T]_\varrho$ and $[\mathbf{1}]_\varrho$ means that `@>` can only be used on functions that accept and return the *same* endpoint. In turn, the fact that the *same* ϱ decorates both

$[T;S]_\varrho$ and $[S]_\varrho$ guarantees that f `@>` u evaluates to the *same* endpoint u that was supplied to f with its type changed to S.

Going back to `stack`, how should we patch its code so that the (inferred) session type of the endpoint accepted by `stack` is S_{none} instead of $S_{\mathtt{reg}}$? We are guided by an easy rule of thumb: place resumptions where the code concludes a sub-protocol of the whole protocol. In this specific case, looking at the protocols (2), we turn the recursive applications `(some x u)` on lines 5 and 10 to `(some x @> u)`, for every interaction with the stack allowed by the recursive call to `some x u` is meant to be concluded by popping `x` from the stack. Thus, using the type system we present in this paper, we obtain a typing derivation proving that the revised `stack` has type $[S_{\mathsf{none}}]_\varrho \to [\mathbf{1}]_\varrho$. Most importantly, the type system makes no use of structural rules or polymorphic recursion and there is no ambiguity as to which protocol `stack` is supposed to carry out, for occurrences of `_;_` in a protocol are tied to the occurrences of `@>` in code that complies with such protocol.

As we will see, these properties make our type system easy to embed in any host programming language supporting parametric polymorphism and (optionally) existential types. This way, we can benefit from an off-the-shelf solution to context-free session type checking and inference instead of developing specific checking/inference algorithms. In the remainder of the paper:

- We formalize a core functional programming language called FuSe$^{\{\}}$ (FuSe with resumptions) that features threads, session-based communication primitives and a distinctive low-level construct for resuming session endpoints (Section 2). The semantics of resumption combinators (including, but not limited to, `@>`) will be explained using this construct.
- We equip FuSe$^{\{\}}$ with an original sub-structural type system that features context-free session types and abstract endpoint identities (Sections 3 and 4). We show that subtyping for context-free session types is undecidable and we prove fundamental properties of well-typed programs emphasizing the implications of these properties in presence of resumptions.
- We detail two `OCaml` modules that implement the distinguishing features of FuSe$^{\{\}}$, namely explicit resumptions and endpoint identities (Section 5). Although a gap remains between FuSe$^{\{\}}$ and the implementations, the modules provide a practical solution to the problems of context-free session type checking and inference, striking different balances among static safety, faithfulness to the formalization of FuSe$^{\{\}}$ and portability.

We defer the discussion of a more elaborate example and of related work to the end of the paper (Sections 6 and 7). Proofs and additional technical material related to Sections 3, 4 and 5 can be found in Appendices A, B and C, respectively. All the code discussed in the paper can be type checked, compiled and run using `OCaml` and both implementations of FuSe$^{\{\}}$. The source code of FuSe [26], which incorporates support for context-free session types, is publicly available [25].

Source of the material. An earlier version of this paper [27] appears in the Proceedings of the 26th European Symposium on Programming (ESOP'17). Unlike [27], the present version adopts a labeled reduction semantics that permits a precise formulation of protocol fidelity (Theorem 4.7). The present version also extends [27] with a more comprehensive set of examples, in-depth discussions on the importance of the uniqueness of endpoint identity, and the full proofs of all the results states in the paper, including the undecidability of context-free session subtyping.

2 A CALCULUS OF FUNCTIONS, SESSIONS AND RESUMPTIONS

The syntax of FuSe$^{\{\}}$ is given in Table 1 and is based on infinite sets of *variables*, *identity variables*, and of *session channels*. We use an involution $\overline{\cdot}$ that turns an identity variable or channel into the (distinct) corresponding identity co-variable or co-channel. Each session channel a has two *endpoints*, one denoted by the channel a itself, the other by the corresponding co-channel $\overline{a}$. We say that a is the *peer endpoint* of $\overline{a}$ and vice versa. Given an endpoint ε, we write $\overline{\varepsilon}$ for its peer. A *name*

Table 1. FuSe$^{\{\}}$: syntax (‡ marks the runtime syntax not used in source programs).

Domains	x, y	$\in$ *Var*	variables
	ϱ	$\in$ *IdVar*	identity variables
	a, b	$\in$ *Channel*	session channels
	ε	$\in$ *Channel* $\cup$ $\overline{\textit{Channel}}$	endpoints
	u	$\in$ *Channel* $\cup$ $\overline{\textit{Channel}}$ $\cup$ *Var*	names
	ι	$\in$ *Channel* $\cup$ $\overline{\textit{Channel}}$ $\cup$ *IdVar* $\cup$ $\overline{\textit{IdVar}}$	identities
	C	$\in$ *Tags*	tags
Processes	P, Q	$::= \langle e \rangle$	thread
		$\mid P \mid Q$	parallel composition‡
		$\mid (\nu a)P$	session‡
Expressions	e	$::= v$	value
		$\mid x$	variable
		$\mid e\ e'$	value application
		$\mid e\ [\iota]$	identity application
		$\mid$ `let` $x, y = e_1$ `in` e_2	pair splitting
		$\mid$ `match` e `with` $\{\mathsf{C}_i \Rightarrow e_i\}_{i \in I}$	pattern matching
		$\mid \lceil \iota, e \rceil$	packing
		$\mid$ `let` $\lceil \varrho, x \rceil = e_1$ `in` e_2	unpacking
		$\mid \{e\}_u$	resumption
Values	v, w	$::= \mathsf{c} \mid (v, w) \mid \mathsf{C}\ v$	data
		$\mid$ `pair` $v \mid$ `fork` $v \mid$ `_send` $v \mid$ `select` $v \mid$ `fix` v	partial application
		$\mid \lambda x.e \mid \Lambda\varrho.v$	abstraction
		$\mid \varepsilon$	endpoint‡
		$\mid \lceil \varepsilon, v \rceil$	package‡
Constants	c	$::=$ `()` $\mid$ `pair` $\mid \mathsf{C} \mid$ `fix` $\mid$ `fork`	
		$\mid$ `create` $\mid$ `_send` $\mid$ `_receive` $\mid$ `select` $\mid$ `branch`	

is either an endpoint or a variable. An *identity* is either an endpoint or an identity (co-)variable. We write $\bar{\iota}$ for the co-identity of ι, which is defined in such a way that $\overline{\bar{\varrho}} = \varrho$.

The syntax of expressions is mostly standard and comprises constants, variables, abstractions, applications, and two forms for splitting pairs and matching tagged values. Constants, ranged over by c, comprise the unitary value `()`, the pair constructor `pair`, an arbitrary set of tags C for tagged unions, the fixpoint operator `fix`, a primitive `fork` for creating new threads, and a set of session primitives [10, 26] whose semantics will be detailed shortly. To improve readability, we write (e_1, e_2) in place of the saturated application `pair` $e_1\ e_2$. In addition, the calculus provides abstraction, application, packing and unpacking of identities. These respectively correspond to introduction and elimination constructs for universal and existential types, which are limited to identities in the formal development of FuSe$^{\{\}}$. The binder symbol of identity abstractions is Λ, to distinguish it from value abstractions. The distinguishing feature of FuSe$^{\{\}}$ is the resumption construct $\{e\}_u$ indicating that e uses the endpoint u for completing some prefix of a sequentially composed protocol. As we will see in Example 4.8, resumptions are keys to define operators such as `@>` introduced in Section 1. Values are fairly standard except for two details that are easy to overlook. First, `fix` v is a value and reduces only when applied to a further argument. This approach, already used by Tov [32], simplifies the operational semantics (and the formal proofs) sparing us the need

Table 2. $\mathsf{FuSe}^{\{\}}$: operational semantics.

Reduction of expressions $\boxed{e \longrightarrow e'}$

[R1]	$(\lambda x.e)\ v \longrightarrow e\{v/x\}$	
[R2]	$(\Lambda\varrho.v)\ [\varepsilon] \longrightarrow v\{\varepsilon/\varrho\}$	
[R3]	$\mathtt{fix}\ v\ w \longrightarrow v\ (\mathtt{fix}\ v)\ w$	
[R4]	$\mathtt{let}\ x, y = (v, w)\ \mathtt{in}\ e \longrightarrow e\{v/x\}\{w/y\}$	
[R5]	$\mathtt{match}\ (\mathsf{C}_k\ v)\ \mathtt{with}\ \{\mathsf{C}_i \Rightarrow e_i\}_{i\in I} \longrightarrow e_k\ v$	$k \in I$
[R6]	$\mathtt{let}\ \lceil\varrho, x\rceil = \lceil\varepsilon, v\rceil\ \mathtt{in}\ e \longrightarrow e\{\varepsilon/\varrho\}\{v/x\}$	
[R7]	$\{(v, \varepsilon)\}_\varepsilon \longrightarrow (v, \varepsilon)$	

Reduction of processes $\boxed{P \xrightarrow{\ell} Q}$

[R8]	$\langle \mathcal{E}[\mathtt{fork}\ v\ w]\rangle \xrightarrow{\tau} \langle \mathcal{E}[()]\rangle \mid \langle v\ w\rangle$	
[R9]	$\langle \mathcal{E}[\mathtt{create}\ ()]\rangle \xrightarrow{\tau} (\nu a)\langle \mathcal{E}[\lceil a, (a, \overline{a})\rceil]\rangle$	$a \notin \mathsf{fn}(\mathcal{E})$
[R10]	$\langle \mathcal{E}[\_\mathtt{send}\ v\ \varepsilon]\rangle \mid \langle \mathcal{E}'[\_\mathtt{receive}\ \overline{\varepsilon}]\rangle \xrightarrow{\varepsilon} \langle \mathcal{E}[\varepsilon]\rangle \mid \langle \mathcal{E}'[(v, \overline{\varepsilon})]\rangle$	
[R11]	$\langle \mathcal{E}[\mathtt{select}\ v\ \varepsilon]\rangle \mid \langle \mathcal{E}'[\mathtt{branch}\ \overline{\varepsilon}]\rangle \xrightarrow{\varepsilon} \langle \mathcal{E}[\varepsilon]\rangle \mid \langle \mathcal{E}'[v\ \overline{\varepsilon}]\rangle$	
[R12]	$\langle \mathcal{E}[e]\rangle \xrightarrow{\tau} \langle \mathcal{E}[e']\rangle$	if $e \longrightarrow e'$
[R13]	$P \mid R \xrightarrow{\ell} Q \mid R$	if $P \xrightarrow{\ell} Q$
[R14]	$(\nu a)P \xrightarrow{\ell} (\nu a)Q$	if $P \xrightarrow{\ell} Q$ and $\ell \notin \{a, \overline{a}\}$
[R15]	$(\nu a)P \xrightarrow{\tau} (\nu a)Q$	if $P \xrightarrow{\ell} Q$ and $\ell \in \{a, \overline{a}\}$
[R16]	$P \xrightarrow{\ell} Q$	if $P \equiv P' \xrightarrow{\ell} Q' \equiv Q$

to η-expand `fix` each time it is unfolded [37]. Second, the body of an identity abstraction $\Lambda\varrho.v$ is a value and not an arbitrary expression. This restriction, inspired by [32, 34], simplifies the type system without affecting expressiveness since the body of an identity abstraction is usually another (identity or value) abstraction. In this respect, the fact that `fix` v is a value allows us to write identity-monomorphic, recursive functions of the form $\Lambda\varrho.\mathtt{fix}\ \lambda f.e$ which are both common and useful in practice. Processes are parallel compositions of threads possibly connected by sessions. The definition of free and bound names for both expressions and processes is the obvious one, except that a restriction $(\nu a)P$ binds both a and $\overline{a}$ in P. We identify terms modulo alpha-renaming of bound names and we write $\mathsf{fn}(e)$ (respectively, $\mathsf{fn}(P)$) for the free names of e (respectively, P).

Table 2 defines the (call-by-value) operational semantics of $\mathsf{FuSe}^{\{\}}$, where we write $e\{v/x\}$ and $e\{\iota/\varrho\}$ for the (capture-avoiding) substitutions of values and identities in place of variables and identity variables, respectively. Evaluation contexts are essentially standard:

$$\begin{array}{lrcl} \textbf{Context} & \mathcal{E} & ::= & [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \lceil \iota, \mathcal{E}\rceil \mid \mathtt{let}\ \lceil\varrho, x\rceil = \mathcal{E}\ \mathtt{in}\ e \mid \{\mathcal{E}\}_u \\ & & \mid & \mathtt{let}\ x, y = \mathcal{E}\ \mathtt{in}\ e \mid \mathtt{match}\ \mathcal{E}\ \mathtt{with}\ \{\mathsf{C}_i \Rightarrow e_i\}_{i\in I} \end{array}$$

Reduction rules [R1–R6] are unremarkable. The reduction rule [R7] erases the resumption $\{\cdot\}_\varepsilon$ around a pair (v, ε), provided that the endpoint in the right component of the pair matches the annotation of the resumption. The type system for $\mathsf{FuSe}^{\{\}}$ that we are going to define enforces this condition statically. However, the rule also suggests an implementation of resumptions based on a simple runtime check: $\{(v, \varepsilon)\}_{\varepsilon'}$ reduces to (v, ε) if ε and ε' are the same endpoint and fails (*e.g.* raising an exception) otherwise. This alternative semantics may be useful if the type system of the host language is not expressive enough to enforce the typing discipline described in Section 3. We will consider this alternative semantics for one of the two implementations of $\mathsf{FuSe}^{\{\}}$ (Section 5.2).

The operational semantics of processes is essentially that given by Thiemann and Vasconcelos [31] and Padovani [26] and is defined in terms of a structural congruence relation $\equiv$ and a labeled reduction relation $\xrightarrow{\ell}$. The relation $\equiv$ is defined as the least congruence such that

$$P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \frac{\{a, \overline{a}\} \cap \mathsf{fn}(Q) = \emptyset}{(\nu a)P \mid Q \equiv (\nu a)(P \mid Q)}$$

The label ℓ in a relation $P \xrightarrow{\ell} Q$ is either an endpoint ε, meaning that (some sub-process of) P sends a message over the endpoint ε, or the special symbol τ, meaning that P evolves into Q by means of an unobservable action. We now discuss each reduction rule. Rule [R8] describes the spawning of a new thread, whose body is the application of `fork`'s arguments. We have chosen this semantics of `fork` so that it matches `OCaml`'s. Rule [R9] models session initiation, whereby `create` reduces to a pair with the two endpoints of the newly created session. The side condition makes sure that the new session does not capture any free name in the thread creating it. Compared to [10], we have one primitive that returns both endpoints of a new session instead of a pair of primitives that synchronize over shared/public channels. This choice is mostly a matter of technical economy: session initiation based on shared/public channels can be programmed on top of this mechanism. Also, the pair returned by `create` is packed to account for the fact that the caller of `create` does not know the identities of the endpoints therein. Note that, in the residual process, the leftmost occurrence of a represents an identity, hence it does not count as an actual usage of the endpoint a. Rule [R10] models communication, moving the message from the sender to the receiver and pairing the message with the continuation endpoint on the receiver side. Asynchronous communication in the style of [10] can be accommodated without affecting any of the results that follow. The primitives `_send` and `_receive`, while having the same operational semantics as the more common `send` and `receive` found in other works [10, 26, 31, 35], are named differently because they will be typed differently (Section 4). The usual `send` and `receive` will be defined in terms of `_send` and `_receive` (Example 4.9). Rule [R11] models selections by applying the first argument of `select` to the receiver's continuation endpoint. Typically, the first argument of `select` will be a tag `C` which is effectively the message being exchanged in this case. We adopt this slightly unusual semantics of `select` because it models accurately the implementation and, at the same time, it calls for specific features of the type system concerning the type-level identification of endpoints. Rule [R12] lifts reductions from expressions to processes and rules [R13–R16] close reductions under parallel compositions, restrictions, and structural congruence.

3 TYPES

Syntax. In this section we define the type language for $\mathtt{FuSe}^{\{\}}$. To keep the formal development as simple as possible and to focus on the distinguishing traits of $\mathtt{FuSe}^{\{\}}$, we limit polymorphism to identity variables. This limitation will be lifted in the actual implementation, where we leverage on `OCaml`'s support for parametric polymorphism. The (finite) syntax of kinds, types, and session types is given below:

$$\begin{array}{rrcl} \textbf{Kind} & \kappa & ::= & \mathsf{U} \mid \mathsf{L} \\ \textbf{Type} & t, s & ::= & \mathtt{unit} \mid t \times s \mid \{\mathtt{C}_i \ \mathtt{of}\ t_i\}_{i \in I} \mid t \to^{\kappa} s \mid [T]_{\iota} \mid \exists \varrho.t \mid \forall \varrho.t \\ \textbf{Session type} & T, S & ::= & \mathbf{0} \mid \mathbf{1} \mid ?t \mid !t \mid \&[\mathtt{C}_i : T_i]_{i \in I} \mid \oplus[\mathtt{C}_i : T_i]_{i \in I} \mid T\,;S \end{array}$$

Instead of introducing concrete syntax for recursive (session) types, we let t, s and T, S range over the possibly infinite, regular trees generated by the above constructors for types and session types, respectively. We introduce recursive (session) types as solutions of finite systems of (session) type equations, such as (1). The shape of the equation, with the metavariable S_{reg} occurring unguarded

on the left hand side and guarded by at least one constructor on the right hand side, guarantees that the equation has exactly one solution [3]. Type equality corresponds to regular tree equality.

The kinds U and L are used to classify types as unlimited and linear, respectively. Types of kind U denote values that can be used any number of times. Types of kind L denote values that must be used exactly once. We introduce a few more notions before seeing how kinds are assigned to types.

Types include a number of *base types* (such as `unit`, `int` and possibly others used in the examples), *products* $t \times s$, and *tagged unions* $\{\mathtt{C}_i \ \mathtt{of}\ t_i\}_{i\in I}$. The *function type* $t \to^\kappa s$ has a kind annotation κ indicating whether the function can be applied any number of times ($\kappa = \mathsf{U}$) or must be applied exactly once ($\kappa = \mathsf{L}$). This latter constraint typically arises when the function contains linear values in its closure. We often omit the annotation κ when it is U. An *endpoint type* $[T]_\iota$ consists of a session type T, describing the protocol according to which the endpoint must be used, and an identity ι of the endpoint. Finally, we have *existential and universal quantifiers* $\exists \varrho.t$ and $\forall \varrho.t$ over identity variables. These are the only binders in types. We write $\mathsf{fid}(t)$ for the set of identities occurring free in t and we identify (session) types modulo renaming of bound identities.

A session type describes the sequence of actions to be peformed on an endpoint. The basic actions $?t$ and $!t$ respectively denote the input and the output of a message of type t. As in [31] and unlike most presentations of session types, these forms do not specify a continuation, which can be attached using sequential composition. External choices $\&[\mathtt{C}_i : T_i]_{i\in I}$ and internal choices $\oplus[\mathtt{C}_i : T_i]_{i\in I}$ describe protocols that can proceed according to different continuations T_i each associated with a tag $\mathtt{C}_i$. When the choice is internal, the process using the endpoint selects the continuation. When the choice is external, the process accepts the selection performed on the peer endpoint. Therefore, an external choice corresponds to an input (of a tag $\mathtt{C}_i$) and an internal choice to an output. Sequential composition $T;S$ combines two sub-protocols T and S into a protocol where all the actions described by T are supposed to be performed before any action described by S. The session type $\mathbf{0}$ describes a terminated protocol whereas $\mathbf{1}$ describes a terminated sub-protocol. As we will see, this distinction affects also the kinding of endpoint types: an endpoint of session type $\mathbf{0}$ can be discarded for it serves no purpose, whereas an endpoint of session type $\mathbf{1}$ must be resumed exactly once.

Session type transitions. We define a *labeled transition system* (LTS) that formalizes the (observable) actions allowed by a session type. This notion is instrumental in defining several key ingredients of the type system. Labels of the LTS represent in an abstract form the message that is either sent ($!$) or received ($?$) and their syntax is defined thus:

$$\begin{array}{rrcl} \textbf{Message Type} & \omega & ::= & t \mid \mathtt{C} \\ \textbf{Label} & \mu & ::= & ?\omega \mid !\omega \end{array}$$

Definition 3.1 (session type LTS). Let $\mathsf{done}(\cdot)$ be the least predicate on session types inductively defined by the following axiom and rule:

$$\mathsf{done}(\mathbf{1}) \qquad \frac{\mathsf{done}(T) \qquad \mathsf{done}(S)}{\mathsf{done}(T;S)}$$

Let $\xrightarrow{\mu}$ be the least family of relations on session types inductively defined by the following axioms and rules:

$$?t \xrightarrow{?t} \mathbf{1} \qquad !t \xrightarrow{!t} \mathbf{1}$$

$$\frac{k \in I}{\&[\mathtt{C}_i : T_i]_{i\in I} \xrightarrow{?\mathtt{C}_k} T_k} \qquad \frac{k \in I}{\oplus[\mathtt{C}_i : T_i]_{i\in I} \xrightarrow{!\mathtt{C}_k} T_k} \qquad \frac{T \xrightarrow{\mu} T'}{T;S \xrightarrow{\mu} T';S} \qquad \frac{\mathsf{done}(T) \qquad S \xrightarrow{\mu} S'}{T;S \xrightarrow{\mu} S'}$$

We write $T \xrightarrow{\mu}$ if there exists S such that $T \xrightarrow{\mu} S$. We now use these notions to characterize all session types that describe terminated protocols:

$$\mathsf{terminated}(T) \stackrel{\text{def}}{\iff} \neg\mathsf{done}(T) \wedge \nexists \mu : T \xrightarrow{\mu}$$

Note that $\mathbf{0}$ is the simplest session type describing a terminated protocol, but is not the only one. For example, $\mathsf{terminated}(T)$ holds also for the session type T that satisfies the equation $T = T\,;T$.

Subtyping and type equivalence. We introduce the notion of type equivalence in two steps. First, we define a subtyping relation $\lesssim$ for session types that is consistent with the conventional one [9]. Then, we define type equivalence $\sim$ as the equivalence relation induced by $\lesssim$. The reason to define equivalence through subtyping (instead of directly as done by Thiemann and Vasconcelos [31]) is to point out a substantial difference between the two relations that affects their adoption in practice.

Definition 3.2 (subtyping for session types). We write $\lesssim$ for the largest binary relation on session types such that $T \lesssim S$ implies either:

(1) $\mathsf{terminated}(T)$ and $\mathsf{terminated}(S)$, or
(2) $\mathsf{done}(T)$ and $\mathsf{done}(S)$, or
(3) $T \xrightarrow{?\omega'}$ and $T \xrightarrow{?\omega} T'$ implies $S \xrightarrow{?\omega} S'$ and $T' \lesssim S'$, or
(4) $S \xrightarrow{!\omega'}$ and $S \xrightarrow{!\omega} S'$ implies $T \xrightarrow{!\omega} T'$ and $T' \lesssim S'$.

We write $T \sim S$ if $T \lesssim S$ and $S \lesssim T$.

It is easy to see that $\lesssim$ is a partial order and that $\sim$ is an equivalence relation. In particular, $\lesssim$ is consistent with the standard notion of subtyping for session types [9]. For example, we have $\oplus[\mathsf{Push} : T, \mathsf{Done} : S] \lesssim \oplus[\mathsf{Push} : T]$ for an endpoint on which both Push and Done can be selected can be safely used in place of an endpoint on which only Push can. Dually, $\&[\mathsf{Push} : T] \lesssim \&[\mathsf{Push} : T, \mathsf{Done} : S]$ for an endpoint from which only Push can be received can be safely used in place of an endpoint from which either Push or Done can. It is also easy to see that $\sim$ coincides with the bisimulation relation defined by Thiemann and Vasconcelos [31]: two equivalent session types allow exactly the same sequences of actions. The relation $\sim$ allows us to rewrite session types according to the expected monoidal and distributive laws for sequential composition. More precisely:

Proposition 3.3 (properties of $\lesssim$). *The following properties hold:*

(1) *(associativity)* $T\,;(S\,;R) \sim (T\,;S)\,;R$;
(2) *(unit)* $\mathbf{1}\,;T \sim T\,;\mathbf{1} \sim T$;
(3) *(distributivity)* $\&[\mathsf{C}_i : T_i]_{i\in I}\,;T \sim \&[\mathsf{C}_i : T_i\,;T]$ *and* $\oplus[\mathsf{C}_i : T_i]_{i\in I}\,;T \sim \oplus[\mathsf{C}_i : T_i\,;T]$;
(4) *(pre-congruence)* $T \lesssim T'$ *and* $S \lesssim S'$ *imply* $T\,;S \lesssim T'\,;S'$.

The pre-congruence property of $\lesssim$ is particularly important in our setting since we use sequential composition as a modular construct for structuring programs. We do *not* identify equivalent session types, and we assume that sequential composition associates to the right: $T\,;S\,;R$ means $T\,;(S\,;R)$.

Oddly enough, the equivalence problem for context-free session types is decidable but subtyping is not. The decidability of equivalence has already been established by Thiemann and Vasconcelos by encoding session types into terms of Basic Process Algebra, for which bisimulation is known to be decidable. Undecidability of subtyping for context-free session types follows from known results in formal language theory [7] which have identified a number of language classes for which language equivalence is decidable and inclusion is not.

Proposition 3.4. *The relation $\sim$ is decidable, whereas $\lesssim$ is not.*

Compared to [31], the decidability of equivalence has a minor relevance in our setting since $\sim$ is never used in the typing rules concerning user syntax. On the other hand, the undecidability of subtyping shows that the increased expressiveness of context-free session types comes at a price: if one tries to integrate subtyping in a context-free session type system, the subtyping relation must be restricted somehow in order to preserve decidability. By tying the sequential structure of session types to the structure of the code, resumptions are one mechanism to implement this restriction. In fact, the implementations we will describe in Section 5 leverage on `OCaml`'s subtyping relation to provide a sound (and necessarily incomplete) approximation of $\lesssim$.

Kinding. We are now ready to classify types according to their kind. We resort to a coinductive definition to cope with possibly infinite types.

Definition 3.5 (kinding). Let $::$ be the largest relation between types and kinds such that $t :: \kappa$ implies either:

- $\kappa = \mathsf{L}$, or
- $t = \mathtt{unit}$, or
- $t = t_1 \to^{\mathsf{U}} t_2$, or
- $t = [T]_\iota$ and $\mathsf{terminated}(T)$, or
- $t = \exists \varrho.s$ and $s :: \kappa$, or
- $t = \forall \varrho.s$ and $s :: \kappa$, or
- $t = t_1 \times t_2$ and $t_1 :: \kappa$ and $t_2 :: \kappa$, or
- $t = \{\mathtt{C}_i \ \mathtt{of}\ t_i\}_{i\in I}$ and $t_i :: \kappa$ for every $i \in I$.

We say that t is *unlimited* if $t :: \mathsf{U}$ and that t is *linear* if its only kind is L, namely if $t :: \kappa$ implies $\kappa = \mathsf{L}$. Endpoint types and function types with kind annotation L are linear since they denote values that must be used exactly once. Base types and function types with kind annotation U are unlimited since they denote values that can be used without limitations or even discarded. Note that the kind of a function type $t \to^\kappa s$ solely depends on κ and not on the kind of t or s. For example, $[\mathtt{?int}]_\iota \to \mathtt{int}$ is unlimited even if $[\mathtt{?int}]_\iota$ is not. An endpoint type $[T]_\iota$ is linear as long as T is not (equivalent to) the terminated session type $\mathbf{0}$. The kind of existential and universal types, products and tagged unions is determined by that of the component types. For example, the type $t = \{\mathtt{Nil}\ \mathtt{of}\ \mathtt{unit}, \mathtt{Cons}\ \mathtt{of}\ \mathtt{int} \times t\}$ of integer lists is unlimited, whereas the type $\mathtt{int} \times [\mathtt{!int}]_\iota$ is linear. Finally, note that every type has kind L and therefore Definition 3.5 accounts for a form of *subkinding*: $t :: \mathsf{U}$ implies $t :: \mathsf{L}$. This is motivated by the observation that it is safe to use a value of an unlimited type exactly once.

Duality. As usual, the session types associated with peer endpoints must be dual to each other to guarantee communication safety. Duality expresses the fact that every input action performed on an endpoint is matched by a corresponding output performed on its peer and is defined thus:

Definition 3.6 (session type duality). *Session type duality* is the function $\overline{\cdot}$ coinductively defined by the following equations:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl@{\qquad}l}
\overline{\mathbf{0}} & = & \mathbf{0} & \overline{?t} & = & !t & \overline{\&[\mathtt{C}_i : T_i]_{i\in I}} & = & \oplus[\mathtt{C}_i : \overline{T}_i]_{i\in I} & \overline{T;S} = \overline{T};\overline{S} \\
\overline{\mathbf{1}} & = & \mathbf{1} & \overline{!t} & = & ?t & \overline{\oplus[\mathtt{C}_i : T_i]_{i\in I}} & = & \&[\mathtt{C}_i : \overline{T}_i]_{i\in I} &
\end{array}$$

It is easy to verify that duality is an involution, that is $\overline{\overline{T}} = T$.

Table 3. Type schemes of FuSe$^{\{\}}$ constants.

$() : \mathtt{unit}$		
$\mathtt{pair} : t \to s \to^{\kappa} t \times s$	$t :: \kappa$	
$\mathtt{C}_j : t_j \to \{\mathtt{C}_i \ \mathtt{of}\ t_i\}_{i \in I}$	$j \in I$	
$\mathtt{fix} : ((t \to s) \to t \to s) \to t \to s$		
$\mathtt{fork} : (t \to \mathtt{unit}) \to t \to \mathtt{unit}$		
$\mathtt{create} : \mathtt{unit} \to \exists \varrho.([T]_{\varrho} \times [\overline{T}]_{\overline{\varrho}})$		
$\mathtt{\_send} : t \to [!t]_{\iota} \to^{\kappa} [\mathbf{1}]_{\iota}$	$t :: \kappa$	
$\mathtt{\_receive} : [?t]_{\iota} \to t \times [\mathbf{1}]_{\iota}$		
$\mathtt{select} : ([\overline{T_j}]_{\overline{\iota}} \to^{\kappa} \{\mathtt{C}_i \ \mathtt{of}\ [\overline{T_i}]_{\overline{\iota}}\}_{i \in I}) \to [\oplus[\mathtt{C}_i : T_i]_{i \in I}]_{\iota} \to^{\kappa} [T_j]_{\iota}$	$j \in I$	
$\mathtt{branch} : [\&[\mathtt{C}_i : T_i]_{i \in I}]_{\iota} \to \{\mathtt{C}_i \ \mathtt{of}\ [T_i]_{\iota}\}_{i \in I}$		

4 TYPE SYSTEM

The type system makes use of two environments: identity environments Δ are sets of identities written $\iota_1, \ldots, \iota_n$, representing the endpoints statically known to a program fragment; type environments Γ are finite maps from names to types written $u_1 : t_1, \ldots, u_n : t_n$ associating a type with every (free) name occurring in an expression. We write Δ, Δ' for $\Delta \cup \Delta'$ when $\Delta \cap \Delta' = \emptyset$. We write $\Gamma(u)$ for the type associated with u in Γ, $\mathrm{dom}(\Gamma)$ for the domain of Γ, and Γ_1, Γ_2 for the union of Γ_1 and Γ_2 when $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. We extend kinding to type environments in the obvious way, writing $\Gamma :: \kappa$ if $\Gamma(u) :: \kappa$ for all $u \in \mathrm{dom}(\Gamma)$. Note that thanks to subkinding $\Gamma :: \mathsf{L}$ holds for all Γ, whereas $\Gamma :: \mathsf{U}$ holds only if all types in the range in Γ are unlimited.

We also need a partial operator to combine type environments with possibly overlapping domains. This operator is common in substructural type systems and allows names with unlimited types to be used any number of times.

Definition 4.1 (environment combination [18]). We write $+$ for the partial operation on type environments such that:

$$\Gamma + \Gamma' \stackrel{\text{def}}{=} \Gamma, \Gamma' \qquad \text{if } \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$$
$$(\Gamma, u : t) + (\Gamma', u : t) \stackrel{\text{def}}{=} (\Gamma + \Gamma'), u : t \qquad \text{if } t :: \mathsf{U}$$

Note that $\Gamma + \Gamma'$ is undefined if Γ and Γ' contain associations for the same name with different or linear types. When $\Gamma :: \mathsf{U}$, we have that $\Gamma + \Gamma$ is always defined and equal to Γ itself.

The type schemes of FuSe$^{\{\}}$ constants are given in Table 3 as associations $\mathtt{c} : t$. Note that, in general, each constant has infinitely many types. Although most associations are as expected, it is worth commenting on a few details. First, observe that the kind annotation κ in the types of `pair`, `_send` and `select` coincides with the kind of the first argument of these constants. In particular, when t is linear and `pair`/`_send`/`select` is supplied one argument v of type t, the resulting partial application is also linear. Second, in accordance with their operational semantics (Table 2) all the primitives for session communications (`_send`, `_receive`, `select`, and `branch`) return the very same endpoint they take as input as indicated by the identity ι that annotates the endpoint types in both the domain and range of these constants. Third, in an application `select` v ε, the function v is meant to be applied to the *peer* of ε. This constraint is indicated by the use of the co-identity $\bar{\iota}$ and is key for proving the soundness of the type system. Note also that the codomain of v matches the return type of `branch`, following the fact that v is applied to the peer of ε *after* the communication has occurred (Table 2). Finally, `create` returns a packaged pair of endpoints with dual session types. The package must be opened before the endpoints can

Table 4. FuSe$^{\{\}}$: static semantics.

Typing rules for expressions $\boxed{\Delta;\Gamma \vdash e : t}$

$$\frac{\Gamma :: \mathsf{U} \qquad \mathsf{c} : t}{\Delta;\Gamma \vdash \mathsf{c} : t}\ [\text{T-CONST}] \qquad \frac{\Delta;\Gamma_1 \vdash e_1 : t_1 \times t_2 \qquad \Delta;\Gamma_2, x : t_1, y : t_2 \vdash e_2 : t}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2 : t}\ [\text{T-SPLIT}]$$

$$\frac{\Gamma :: \mathsf{U}}{\Delta;\Gamma, u : t \vdash u : t}\ [\text{T-NAME}] \qquad \frac{\Delta;\Gamma_1 \vdash e : \{\mathsf{C}_i\ \mathtt{of}\ t_i\}_{i\in I} \qquad \Delta;\Gamma_2 \vdash e_i : t_i \to^{\kappa_i} t\ {}^{(i\in I)}}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathtt{match}\ e\ \mathtt{with}\ \{\mathsf{C}_i \Rightarrow e_i\}_{i\in I} : t}\ [\text{T-CASE}]$$

$$\frac{\Delta;\Gamma, x : t \vdash e : s \qquad \Gamma :: \kappa}{\Delta;\Gamma \vdash \lambda x.e : t \to^{\kappa} s}\ [\text{T-FUN}] \qquad \frac{\Delta;\Gamma_1 \vdash e_1 : t \to^{\kappa} s \qquad \Delta;\Gamma_2 \vdash e_2 : t}{\Delta;\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : s}\ [\text{T-APP}]$$

$$\frac{\Delta, \varrho;\Gamma \vdash v : t}{\Delta;\Gamma \vdash \Lambda\varrho.v : \forall\varrho.t}\ [\text{T-ID-FUN}] \qquad \frac{\Delta;\Gamma \vdash e : \forall\varrho.t \qquad \iota \in \Delta}{\Delta;\Gamma \vdash e\ [\iota] : t\{\iota/\varrho\}}\ [\text{T-ID-APP}] \qquad \frac{\Delta;\Gamma, u : [T]_\iota \vdash e : t \times [\mathbf{1}]_\iota}{\Delta;\Gamma, u : [T;S]_\iota \vdash \{e\}_u : t \times [S]_\iota}\ [\text{T-RESUME}]$$

$$\frac{\Delta;\Gamma \vdash e : t\{\iota/\varrho\} \qquad \iota \in \Delta}{\Delta;\Gamma \vdash \lceil \iota, e \rceil : \exists\varrho.t}\ [\text{T-PACK}] \qquad \frac{\Delta;\Gamma_1 \vdash e_1 : \exists\varrho.t \qquad \Delta, \varrho;\Gamma_2, x : t \vdash e_2 : s}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathtt{let}\ \lceil \varrho, x \rceil = e_1\ \mathtt{in}\ e_2 : s}\ [\text{T-UNPACK}]$$

Typing rules for processes $\boxed{\Delta;\Gamma \vdash P}$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{unit}}{\Delta;\Gamma \vdash \langle e \rangle}\ [\text{T-THREAD}] \qquad \frac{\Delta;\Gamma_i \vdash P_i\ {}^{(i=1,2)}}{\Delta;\Gamma_1 + \Gamma_2 \vdash P_1 \mid P_2}\ [\text{T-PAR}] \qquad \frac{\Delta, a, \overline{a};\Gamma, a : [T]_a, \overline{a} : [S]_{\overline{a}} \vdash P \qquad T \sim \overline{S}}{\Delta;\Gamma \vdash (\boldsymbol{\nu} a)P}\ [\text{T-SESSION}]$$

be used for communication. As we will see when discussing the typing rules, the type of `create` could be safely relaxed to $\mathtt{unit} \to \exists\varrho.([T]_\varrho \times [S]_{\overline{\varrho}})$ provided that $T \sim \overline{S}$. This would account for the possibility that the two endpoints of a session are used according to session types that are dual to each other up to type equivalence.

The typing rules for FuSe$^{\{\}}$ are given in Table 4 and are used to derive judgments of the form $\Delta;\Gamma \vdash e : t$ for expressions and judgments of the form $\Delta;\Gamma \vdash P$ for processes. A judgment is well formed if all the identities occurring free in Γ and t are included in Δ. From now on we make the implicit assumption that all judgments are well formed.

We now discuss the most important aspects of the typing rules. In [T-CONST], the implict well-formedness constraint on typing judgments restricts the set of types that we can give to a constant to those whose free identities occur in Δ. In [T-CONST] and [T-NAME], the unused part of the type environment must be unlimited, to make sure that no linear name is left unused. The elimination rules for products and tagged unions are standard. Note the use of + for combining type environments so that the same linear resource is not used multiple times in different parts of an expression. Rules [T-FUN] and [T-APP] deal with function types. In [T-FUN], the kind annotation on the arrow must be consistent with the kind of the environment in which the function is typed. If any name in the environment

has a linear type, then the function must be linear itself to avoid repeated use of such name. By contrast, the kind annotation plays no role in [T-APP]. Abstraction and application of identities are standard. The premise $\iota \in \Delta$ of [T-ID-APP] makes sure that the supplied identity is in scope. This condition is not necessarily captured by the well formedness of judgments in case ϱ does not occur in t. Packing and unpacking are also standard. The identity variable ϱ introduced in [T-UNPACK] is different from any other identity known to e_2. This prevents e_2 from using ϱ in any context where a specific identity is required. Also, well formedness of judgments requires $\mathsf{fid}(s) \subseteq \Delta$, meaning that ϱ is not allowed to escape its scope. The most interesting and distinguishing typing rule of $\mathsf{FuSe}^{\{\}}$ is [T-RESUME]. Let us discuss the rule clockwise, starting from $\{e\}_u$ and recalling that the purpose of this expression is to resume u once the evaluation of e is completed. The rule requires u to have a type of the form $[T\,;S]_\iota$, which specifies the identity ι of the endpoint and the protocols T and S to be completed in this order. Within e the type of u is changed to $[T]_\iota$ and the evaluation of e must yield a pair whose first component, of type t, is the result of the computation and whose second component, of type $[\mathbf{1}]_\iota$, witnesses the fact that the prefix protocol T has been entirely carried out on u. Once the evaluation of e is completed, the type of the endpoint in the pair is reset to the suffix S. The same identity ι relates all the occurrences of the endpoint both in the type environments and in the expressions. Note that the annotation u in $\{\,\cdot\,\}_u$ does not count as a proper "use" of u. Its purpose is solely to identify the endpoint being resumed.

The typing rules for processes are mostly unremarkable. In [T-SESSION] the two peers of a session are introduced both in the type environment and in the identity environment. The protocols T and S of peer endpoints are required to be dual to each other modulo session type equivalence. The use of $\sim$ accounts for the possibility that sequential compositions may be arranged differently in the threads using the two peers. For instance, one thread might be using an endpoint with protocol T, and its peer could have type $\mathbf{1}\,;\overline{T}$ in a thread that has not resumed it yet. Still, $T \sim \overline{\mathbf{1}\,;\overline{T}} = \mathbf{1}\,;T$.

We state a few basic properties of the typing discipline focusing on those more closely related to resumptions. To begin with, we characterize the type environments in which expressions and processes without free variables reduce.

Definition 4.2. We say that Γ is *ground* if $\mathrm{dom}(\Gamma)$ contains endpoints only; that it is *well formed* if $\varepsilon \in \mathrm{dom}(\Gamma)$ implies $\Gamma(\varepsilon) = [T]_\varepsilon$; that it is *balanced* if $\varepsilon, \overline{\varepsilon} \in \mathrm{dom}(\Gamma)$ implies $\Gamma(\varepsilon) = [T]_\varepsilon$ and $\Gamma(\overline{\varepsilon}) = [S]_{\overline{\varepsilon}}$ and $T \sim \overline{S}$.

Note that in a well-formed environment the type associated with endpoint ε is annotated with the correct identity of ε, that is ε itself.

As usual for session type systems, we must take into account the possibility that the type associated with session endpoints changes over time. Normally this only happens when processes use endpoints for communications. In our case, however, expressions may also change endpoint types because of resumptions. In order to track these changes, we introduce two relations that characterize the evolution of type environments alongside expressions and processes. The first relation is the obvious extension of equivalence $\sim$ to type environments:

Definition 4.3 (equivalent type environments). Let $\Gamma = \{\varepsilon_i : [T_i]_{\varepsilon_i}\}_{i\in I}$ and $\Gamma' = \{\varepsilon_i : [S_i]_{\varepsilon_i}\}_{i\in I}$. We write $\Gamma \sim \Gamma'$ if $T_i \sim S_i$ for every $i \in I$.

The second relation includes $\sim$ and mimics communications at the type level. As in the operational semantics of $\mathsf{FuSe}^{\{\}}$, we label the relation to keep track of the sender of the message. This will allow us to formally state a protocol fidelity result for $\mathsf{FuSe}^{\{\}}$.

Definition 4.4. Let $\overset{\ell}{\longmapsto}$ be the least family of relations between type environments such that:

$$\begin{array}{rcll} \Gamma & \overset{\tau}{\longmapsto} & \Gamma' & \text{if } \Gamma \sim \Gamma' \\ \Gamma, \varepsilon : [T]_\varepsilon, \overline{\varepsilon} : [S]_{\overline{\varepsilon}} & \overset{\varepsilon}{\longmapsto} & \Gamma, \varepsilon : [T']_\varepsilon, \overline{\varepsilon} : [S']_{\overline{\varepsilon}} & \text{if } T \overset{!\omega}{\longrightarrow} T' \text{ and } S \overset{?\omega}{\longrightarrow} S' \end{array}$$

We write $\overset{\tau}{\Longmapsto}$ for the reflexive, transitive closure of $\overset{\tau}{\longmapsto}$ and $\overset{\varepsilon}{\Longmapsto}$ for the composition $\overset{\tau}{\Longmapsto}\overset{\varepsilon}{\longmapsto}\overset{\tau}{\Longmapsto}$. Concerning subject reduction for expressions, we have:

Theorem 4.5 (subject reduction for expressions). *Let $\Delta; \Gamma \vdash e : t$ where Γ is ground and well formed. If $e \longrightarrow e'$, then $\Delta; \Gamma' \vdash e' : t$ for some Γ' such that $\Gamma \sim \Gamma'$.*

Theorem 4.5 guarantees that resumptions in well-typed programs do not change arbitrarily the session types of endpoints. The only permitted changes are those allowed by session type equivalence. Concerning progress, we have:

Theorem 4.6 (progress for expressions). *If Γ is ground and well formed and $\Delta; \Gamma \vdash e : t$ and $e \not\longrightarrow$, then either e is a value or $e = \mathcal{E}[K\ v]$ for some $\mathcal{E}$, v, w, and K generated by the grammar*

$$K ::= \texttt{fork}\ w \mid \texttt{create} \mid \texttt{\_send}\ w \mid \texttt{\_receive} \mid \texttt{select}\ w \mid \texttt{branch}$$

That is, an irreducible expression that is not a value is a term that is meant to reduce at the level of processes. Note that a resumption $\{(v, \varepsilon)\}_{\varepsilon'}$ is *not* a value and is meant to reduce at the level of expressions via [R7]. Hence, Theorem 4.6 guarantees that in a well-typed program all such resumptions are such that $\varepsilon = \varepsilon'$. An alternative reading for this observation is that each endpoint is guaranteed to have a unique identity in every well-typed program.

Theorem 4.7 (subject reduction for processes). *Let $\Delta; \Gamma \vdash P$ where Γ is ground, well formed and balanced. If $P \overset{\ell}{\longrightarrow} Q$, then $\Delta; \Gamma' \vdash Q$ for some Γ' such that $\Gamma \overset{\ell}{\Longmapsto} \Gamma'$.*

Apart from being a fundamental sanity check for the type system, Theorem 4.7 states that the communications occurring in processes are precisely those permitted by the session types in the type environments. Therefore, Theorem 4.7 gives us a guarantee of protocol fidelity. A particular instance of protocol fidelity concerns sequential composition: a well-typed process using an endpoint with type $T; S$ is guaranteed to perform the actions described by T first, and then those described by S. Other standard properties including communication safety and (partial) progress for processes can also be proved [26].

Example 4.8 (resumption combinators). In prospect of devising an implementation of $\mathsf{FuSe}^{\{\}}$ in an ordinary programming language, the resumption expression $\{\cdot\}_u$ is challenging to deal with: its typing rule involves a non-trivial manipulation of the type environment whereby the type of u changes as u flows into and out of the expression. In practice, it makes sense to encapsulate $\{\cdot\}_u$ expressions in two combinators that can be easily implemented as higher-order functions (Sections 5.2 and 5.3):

$$\begin{array}{lcl} \texttt{@=} & \overset{\text{def}}{=} & \lambda f.\lambda x.\{f\ x\}_x \\ \texttt{@>} & \overset{\text{def}}{=} & \lambda f.\lambda x.\texttt{let}\ \_, x = \{((), f\ x)\}_x\ \texttt{in}\ x \end{array}$$

The combinator @= is a general version of @> that applies to functions returning an actual result in addition to the endpoint to be resumed. We derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta; f : [T]_\iota \to^\kappa t \times [\mathbf{1}]_\iota \vdash f : [T]_\iota \to^\kappa t \times [\mathbf{1}]_\iota}\,[\text{T-NAME}] \quad \dfrac{}{\Delta; x : [T]_\iota \vdash x : [T]_\iota}\,[\text{T-NAME}]}{\Delta; f : [T]_\iota \to^\kappa t \times [\mathbf{1}]_\iota, x : [T]_\iota \vdash f\ x : t \times [\mathbf{1}]_\iota}\,[\text{T-APP}]}{\Delta; f : [T]_\iota \to^\kappa t \times [\mathbf{1}]_\iota, x : [T;S]_\iota \vdash \{f\ x\}_x : t \times [S]_\iota}\,[\text{T-RESUME}]}{\Delta; f : [T]_\iota \to^\kappa t \times [\mathbf{1}]_\iota \vdash \lambda x.\{f\ x\}_x : [T;S]_\iota \to^\kappa t \times [S]_\iota}\,[\text{T-FUN}]}{\Delta; \emptyset \vdash \lambda f.\lambda x.\{f\ x\}_x : ([T]_\iota \to^\kappa t \times [\mathbf{1}]_\iota) \to [T;S]_\iota \to^\kappa t \times [S]_\iota}\,[\text{T-FUN}]$$

for every T, ι, t and S such that $\mathsf{fid}(T) \cup \mathsf{fid}(t) \cup \mathsf{fid}(S) \cup \{\iota\} \subseteq \Delta$. A similar derivation allows us to derive

$$\Delta; \emptyset \vdash \texttt{@>} : ([T]_\iota \to^\kappa [\mathbf{1}]_\iota) \to [T;S]_\iota \to^\kappa [S]_\iota$$

In the implementation, we will give @= and @> their most general type by leveraging OCaml's support for parametric polymorphism. Other combinators for resuming two or more endpoints can be defined similarly. For example,

$$\texttt{@@>} \stackrel{\text{def}}{=} \lambda f.\lambda x.\lambda y.\texttt{let}\ y, x = \{\texttt{let}\ x, y = \{f\ x\ y\}_y\ \texttt{in}\ (y, x)\}_x\ \texttt{in}\ (x, y)$$

is analogous to @>, but resumes two endpoints at once.

Note that an alternative version of FuSe$^{\{\}}$ without the resumption expression $\{\cdot\}_u$ where @= and @> are taken as primitives is not easy to formalize. This is because the body of the function that consumes the prefix of a sequentially composed protocol $T;S$ requires, in general, an arbitrary number of reduction steps during which the session type of the involved endpoint must be split into the prefix T and the suffix S. The library we describe in Section 5 does indeed implement @= and @> as if they were primitives, but does so using unsafe casts. ■

Example 4.9 (communication API). The type of the communication primitives _send and _receive (Table 3) differs from that of the homonymous primitives in standard presentations of session-based functional calculi [10, 26, 31, 35], where the endpoint returned by the primitive can be used immediately for subsequent communications. In particular, standard presentations of GV-like languages provide send and receive primitives that have the same reduction semantics as _send and _receive (Table 2) but types

$$\begin{array}{rcll} \texttt{send} & : & t \to [!t;T]_\iota \to^\kappa [T]_\iota & t :: \kappa \\ \texttt{receive} & : & [?t;T]_\iota \to t \times [T]_\iota & \end{array}$$

We find a communication API based on _send and _receive appealing for its clean correspondence between primitives and session type constructors. In particular, with this API the resumption combinators account for *all* occurrences of _;_ in protocols. Starting from _send and _receive, send and receive can be easily obtained with the help of @= and @>, used below in infix notation:

$$\begin{array}{rcl} \texttt{send} & \stackrel{\text{def}}{=} & \lambda z.\lambda x.\texttt{\_send}\ z\ \texttt{@>}\ x \\ \texttt{receive} & \stackrel{\text{def}}{=} & \lambda x.\texttt{\_receive}\ \texttt{@=}\ x \end{array}$$

Hereafter we will use and implement whichever version of the communication primitives is convenient depending on the context. ■

We devote the last two examples of this section to motivate the distinguishing features of the type system. In particular, we discuss functions that would be well typed (in possibly weaker versions of the type system) and would allow dangerous rearrangements in the session types of endpoints if

the reduction rule [R7] did not require the resumed endpoint to coincide with the annotation of the resumption.

Example 4.10. In this example we illustrate the key role of endpoint identities for the soundness of resumptions. More specifically, we assume to work with the following relaxed version of the [T-RESUME] typing rule

$$\frac{\Delta;\Gamma,u:[T]_{\iota_1}\vdash e:t\times[\mathbf{1}]_{\iota_2}}{\Delta;\Gamma,u:[T;S]_{\iota_1}\vdash\{e\}_u:t\times[S]_{\iota_2}}\ [\text{T-RESUME-RELAXED}]$$

which *does not* require the resumed endpoint to have the same identity as the annotation of the resumption. This relaxed typing could be exploited by the function

$$\mathtt{bad\_split}\stackrel{\text{def}}{=}\lambda x.\lambda y.\{(x,y)\}_x$$

to disrupt the session types of its endpoint arguments. Indeed we could derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x:[T]_{\iota_1},y:[\mathbf{1}]_{\iota_2}\vdash(x,y):[T]_{\iota_1}\times[\mathbf{1}]_{\iota_2}}\ [\text{T-RESUME-RELAXED}]}{x:[T;S]_{\iota_1},y:[\mathbf{1}]_{\iota_2}\vdash\{(x,y)\}_x:[T]_{\iota_1}\times[S]_{\iota_2}}\ [\text{T-FUN}]}{x:[T;S]_{\iota_1}\vdash\lambda y.\{(x,y)\}_x:[\mathbf{1}]_{\iota_2}\to^{\mathsf{L}}[T]_{\iota_1}\times[S]_{\iota_2}}\ [\text{T-FUN}]}{\vdash\lambda x.\lambda y.\{(x,y)\}_x:[T;S]_{\iota_1}\to[\mathbf{1}]_{\iota_2}\to^{\mathsf{L}}[T]_{\iota_1}\times[S]_{\iota_2}}\ [\text{T-FUN}]$$

meaning that `bad_split` could be used to cast the session type of x from $T;S$ to T and the session type of y from $\mathbf{1}$ to S. The first cast could compromise progress since x would be used to carry out only a prefix of its intended protocol. The second cast, on the other hand, could also compromise communication safety as the peer of y could be cast to another session type R unrelated to $\overline{S}$. ■

Example 4.11. This example presents a well-typed function that "swaps" the session types of its endpoint arguments assuming that they have the same identity. The function

$$\mathtt{bad\_swap}\stackrel{\text{def}}{=}\lambda x.\lambda y.\{\mathtt{let}\ \hat{y},\hat{x}=\{(y,x)\}_y\ \mathtt{in}\ (\hat{x},\hat{y})\}_x$$

applied to two endpoints x and y whose types are $[\mathbf{1};T]_\iota$ and $[\mathbf{1};S]_\iota$ respectively, returns a pair containing the same two endpoints, but with their types changed to $[S]_\iota$ and $[T]_\iota$. The derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x:[\mathbf{1}]_\iota,y:[\mathbf{1}]_\iota\vdash(y,x):[\mathbf{1}]_\iota\times[\mathbf{1}]_\iota}}{x:[\mathbf{1}]_\iota,y:[\mathbf{1};S]_\iota\vdash\{(y,x)\}_y:[\mathbf{1}]_\iota\times[S]_\iota}\ [\text{T-RESUME}]\qquad\dfrac{\vdots}{\hat{x}:[S]_\iota,\hat{y}:[\mathbf{1}]_\iota\vdash(\hat{x},\hat{y}):[S]_\iota\times[\mathbf{1}]_\iota}}{x:[\mathbf{1}]_\iota,y:[\mathbf{1};S]_\iota\vdash\mathtt{let}\ \hat{y},\hat{x}=\{(y,x)\}_y\ \mathtt{in}\ (\hat{x},\hat{y}):[S]_\iota\times[\mathbf{1}]_\iota}\ [\text{T-SPLIT}]}{x:[\mathbf{1};T]_\iota,y:[\mathbf{1};S]_\iota\vdash\{\mathtt{let}\ \hat{y},\hat{x}=\{(y,x)\}_y\ \mathtt{in}\ (\hat{x},\hat{y})\}_x:[S]_\iota\times[T]_\iota}\ [\text{T-RESUME}]}{x:[\mathbf{1};T]_\iota\vdash\lambda y.\{\mathtt{let}\ \hat{y},\hat{x}=\{(y,x)\}_y\ \mathtt{in}\ (\hat{x},\hat{y})\}_x:[\mathbf{1};S]_\iota\to^{\mathsf{L}}[S]_\iota\times[T]_\iota}\ [\text{T-FUN}]}{\vdash\lambda x.\lambda y.\{\mathtt{let}\ \hat{y},\hat{x}=\{(y,x)\}_y\ \mathtt{in}\ (\hat{x},\hat{y})\}_x:[\mathbf{1};T]_\iota\to[\mathbf{1};S]_\iota\to^{\mathsf{L}}[S]_\iota\times[T]_\iota}\ [\text{T-FUN}]$$

shows that `bad_swap` is well typed. If there existed two endpoints ε_1 and ε_2 with the same identity ι from two different sessions, `bad_swap` could be used to exchange their protocols, almost certainly causing communication errors in the rest of the computation.

The same function `bad_swap` illustrates the importance of distinguishing the identities of peer endpoints. If ε and $\overline{\varepsilon}$ were given the same identity, then `bad_swap` could be used to swap their roles.

In this case, communication safety would still be guaranteed by the condition $T \sim \overline{S}$ of [T-SESSION] because $\sim$ is symmetric and duality is an involution. However, protocol fidelity (Theorem 4.7) could be violated. ■

5 CONTEXT-FREE SESSION TYPES IN OCAML

In this section we detail two different implementations of FuSe$^{\{\}}$ communication and resumption primitives as OCaml functions. We define a few basic data structures and a convenient OCaml representation of session types (Section 5.1) before describing the actual implementations. The first one (Section 5.2) is easily portable to any programming language supporting parametric polymorphism, but relies on lightweight runtime checks to verify when an endpoint can be safely resumed. The second implementation (Section 5.3) follows more closely the typing discipline of FuSe$^{\{\}}$ presented in Section 3, but relies on advanced features (existential types) of the host language. The particular implementation we describe is based on OCaml's first-class modules [8, 38].

5.1 Basic Setup

To begin with, we define a module Channel that implements *unsafe* communication channels. In turn, Channel is based on OCaml's Event module, which implements communication primitives in the style of Concurrent ML [29].

```
module Channel : sig
  type t
  val create  : unit → t      (* create a new unsafe channel *)
  val send    : α → t → unit (* send a message of type α    *)
  val receive : t → α         (* receive a message of type α *)
end = struct
  type t       = unit Event.channel
  let create   = Event.new_channel
  let send x u = Event.sync (Event.send u (Obj.magic x))
  let receive u = Obj.magic (Event.sync (Event.receive u))
end
```

An unsafe channel is just an Event.channel for exchanging messages of type **unit**. The **unit** type parameter is a placeholder, for communication primitives perform unsafe casts (with Obj.magic) on every exchanged message. Note that Event.send and Event.receive create synchronization events, and communication only happens when these events are passed to Event.sync. Using Event channels is convenient but not mandatory: the rest of our implementation is essentially independent of the underlying communication framework.

The second ingredient of our library is an implementation of *atomic boolean flags*. Since OCaml's type system is not substructural, we are unable to distinguish between linear and unlimited types and, in particular, we are unable to prevent multiple endpoint usages solely using the type system. Following ideas of Tov and Pucella [33] and Hu and Yoshida [13] and the design of FuSe [26], the idea is to associate each endpoint with a boolean flag indicating whether the endpoint can be safely used or not. The flag is initially set to **true**, indicating that the endpoint can be used, and is tested by every operation that uses the endpoint. If the flag is still **true**, then the endpoint can be used and the flag is reset to **false**. If the flag is **false**, then the endpoint has already been used in the past and the operation aborts raising an exception. Atomicity is needed to make sure that the flag is tested and updated in a consistent way in case multiple threads try to use the same endpoint simultaneously.

```
module Flag : sig
  type t
  val create : unit → t (* create a new atomic boolean flag *)
  val use    : t → unit (* mark as used or raise exception  *)
end = struct
  type t     = Mutex.t
  let create = Mutex.create
  let use f  = if not (Mutex.try_lock f) then raise Error
end
```

We represent an atomic boolean flag as a `Mutex.t`, that is a lock in `OCaml`'s standard library. The value of the flag is the state of the mutex: when the mutex is unlocked, the flag is **`true`**. Using the flag means attempting to acquire the lock with the non-blocking function `Mutex.try_lock`. As for `Event` channels, the mutex is a choice of convenience rather than necessity. More lightweight realizations can be considered making some assumptions on the runtime environment [26].

We conclude the setup phase by defining a bunch of `OCaml` singleton types in correspondence with the session type constructors:

```
  type 0          = End      (* terminated protocol     *)
  type 1          = Done     (* terminated sub-protocol *)
  type φ msg      = Message  (* either ?φ or !φ         *)
  type φ tag      = Tag      (* either &[φ] or ⊕[φ]     *)
  type (α,β) seq = Sequence (* α;β                     *)
```

The type parameter φ is the type of the exchanged message in `msg` and a polymorphic variant type representing the available choices in `tag`. The type parameters α and β in `seq` stand for the prefix and suffix protocols of a sequential composition $\alpha;\beta$. The data constructors of these types are never used and are given only because `OCaml` is more liberal in the construction of recursive types when these are concrete rather than abstract. Hereafter, we use τ_1, τ_2, ... to range over `OCaml` types and α, β, ..., φ to range over `OCaml` type variables. Considering that `OCaml` supports equi-recursive types, we ignore once again the concrete syntax for expressing infinite session types and work with infinite trees instead. `OCaml` uses the notation τ `as` α for denoting a type τ in which occurrences of α stand for the type as a whole.

5.2 A Dynamically Checked, Portable Implementation

The first implementation of the library that we present ignores identities in types and verifies the soundness of resumptions by means of a runtime check. In this case, an endpoint type $[T]_\iota$ is encoded as the `OCaml` type (τ_1,τ_2) `t` where τ_1 and τ_2 are determined as follows:

- when T is either **0** or **1**, then both τ_1 and τ_2 are **0** or **1**, respectively;
- when T is $T_1;T_2$, then τ_1 is (T_1,T_2) `seq` and τ_2 its dual $(\overline{T_1},\overline{T_2})$ `seq`;
- when T is an input (either $?t$ or $\&[\mathtt{C}_i : T_i]_{i\in I}$), then τ_1 is the encoding of the received message/choice (either t `msg` or $\{\mathtt{C}_i \texttt{ of } T_i\}_{i\in I}$ `tag`, respectively) and τ_2 is **0**;
- when T is an output (either $!t$ or $\oplus[\mathtt{C}_i : T_i]_{i\in I}$), then τ_1 is **0** and τ_2 is the encoding of the sent message/choice, along the lines of the previous case.

More precisely, types and session types are encoded thus:

Definition 5.1 (encoding of types and session types). Let $[\![\cdot]\!]$ and $\langle\!\langle\cdot\rangle\!\rangle$ be the encoding functions coinductively and mutually defined by the following equations:

$$\begin{aligned}
[\![\mathbf{0}]\!] &= \texttt{(0,0) t} \\
[\![\mathbf{1}]\!] &= \texttt{(1,1) t} \\
[\![T;S]\!] &= \texttt{(([}\![T]\!]\texttt{,[}\![S]\!]\texttt{) seq,([}\![\overline{T}]\!]\texttt{,[}\![\overline{S}]\!]\texttt{) seq) t} \\
[\![?t]\!] &= \texttt{(}\langle\!\langle t\rangle\!\rangle\texttt{ msg,0) t} \\
[\![!t]\!] &= \texttt{(0,}\langle\!\langle t\rangle\!\rangle\texttt{ msg) t} \\
[\![\&[\mathtt{C}_i : T_i]_{i\in I}]\!] &= \texttt{({C}}_i\texttt{ of [}\![T_i]\!]\texttt{}_{i\in I}\texttt{ tag,0) t} \\
[\![\oplus[\mathtt{C}_i : T_i]_{i\in I}]\!] &= \texttt{(0,{C}}_i\texttt{ of [}\![\overline{T_i}]\!]\texttt{}_{i\in I}\texttt{ tag) t} \\
\\
\langle\!\langle[T]_\iota\rangle\!\rangle &= [\![T]\!]
\end{aligned}$$

where $\langle\!\langle\cdot\rangle\!\rangle$ is extended homomorphically to all the remaining type constructors erasing kind annotations on arrows and existential and universal quantifiers.

Note that identities ι in endpoint types are simply erased; we will revise this choice in the second implementation (Section 5.3). The encoding is semantically grounded through the relationship between sessions and linear channels [4, 5, 17, 26] and is extended here to sequential composition. The distinguishing feature of this encoding is that is makes it easy to express session type duality constraints solely in terms of type equality:

Theorem 5.2. *If* $[\![T]\!] = \texttt{(}\tau_1\texttt{,}\tau_2\texttt{) t}$, *then* $[\![\overline{T}]\!] = \texttt{(}\tau_2\texttt{,}\tau_1\texttt{) t}$.

That is, we pass from a session type to its dual by flipping the type parameters of the `t` type. This also works for unknown or partially known session types: the dual of `(α,β) t` is `(β,α) t`.

We can now look at the concrete representation of the type `(α,β) t`:

```
type (α,β) t = { chan : Channel.t; pol : int; once : Flag.t }
```

An endpoint is a record with three fields, a reference `chan` to the unsafe channel used for the actual communications, an integer number `pol` $\in \{+1, -1\}$ representing the endpoint's polarity, and an atomic boolean flag `once` indicating whether the endpoint can be safely used or not. Of course, this representation is hidden from the user of the library and any direct access to these fields occurs via one of the public functions that we are going to discuss. The use of integers for representing polarities is not essential. Any other data type with (at least) two distinct values would be equally fine.

The FuSe$^{\{\}}$ primitives for session communication are implemented by corresponding `OCaml` functions with the following signatures, which are directly related to the type schemes in Table 3 through the encoding in Definition 5.1:

```
val create   : unit → (α,β) t × (β,α) t
val _send    : φ → (0,φ msg) t → (1,1) t
val _receive : (φ msg,0) t → φ × (1,1) t
val select   : ((β,α) t → φ) → (0,φ tag) t → (α,β) t
val branch   : (φ tag,0) t → φ
val (@=)     : ((α,β) t → φ × (1,1) t) →
               (((α,β) t,(γ,δ) t) seq,((β,α) t,(δ,γ) t) seq) t → φ × (γ,δ) t
```

We take advantage of parametric polymorphism to give these functions their most general types. We omit the type declaration of `@>` which is just a particular instance of `@=` (Example 4.8). The types for `select` and `branch` are slightly more general than those in Table 3, but the passing of

tags between choices and unions cannot be expressed as accurately in OCaml without fixing the set of tags. The given typing is still sound though.

Since this version of the library ignores endpoint identities, the endpoints returned by create are already unpackaged. The implementation of create is

```
let create () = let ch = Channel.create () in
                { chan = ch; pol = +1; once = Flag.create () },
                { chan = ch; pol = -1; once = Flag.create () }
```

and consists of the creation of a new unsafe channel ch and two records referring to it with opposite polarities each with its own validity flag.

The communication primitives are defined in terms of corresponding operations on the underlying unsafe channel and make use of an auxiliary function

```
let fresh u = { u with once = Flag.create () }
```

that returns a copy of u with once overwritten by a fresh flag. We have:

```
let _send x u  = Flag.use u.once; Channel.send x u.chan; fresh u
let _receive u = Flag.use u.once; (Channel.receive u.chan, fresh u)
let select f u = Flag.use u.once; Channel.send f u.chan; fresh u
let branch u   = Flag.use u.once; Channel.receive u.chan (fresh u)
```

The flag associated with the endpoint u is used before communication takes place and refreshed just before the endpoint is returned to the user. It is not possible to refresh the flag by just releasing the lock in it, for any existing alias to the endpoint must be permanently marked as invalid [26].

We complete the module with the implementation of @=, shown below:

```
let (@=) scope u =
  let res, v = scope (Obj.magic u) in
  if u.chan == v.chan && u.pol = v.pol then (res, Obj.magic v)
  else raise Error
```

The endpoint u is passed to scope, which evaluates to a pair made of the result res of the computation and the endpoint v to be resumed. The cast Obj.magic u is necessary to turn the type of u from $T;S$ to T, as required by scope. The second line in the body of @= checks that the endpoint v resulting from the evaluation of scope is indeed the same endpoint u that was fed in it. Note the key role of the polarity in checking that u and v are the same endpoint and the use of the physical equality operator ==, which compares only the *references* to the involved unsafe channels. An exception is raised if v is not the same endpoint as u. Otherwise, the result of the computation and v are returned. The cast Obj.magic v effectively resumes the endpoint turning its type from $\mathbf{1}$ to S. The two casts roughly delimit the region of code that we would write within $\{\cdot\}_u$ in the formal model of FuSe$^{\{\}}$.

The implementation of @> follows immediately from its definition as by Example 4.8:

```
let (@>) scope u = snd ((fun u → ((), scope u)) @= u)
```

Example 5.3. In this example we attempt to break the soundness of the resumption operator by implementing an escape function of type $[\mathbf{1};T]_\varrho \to [T]_\sigma$ that changes the identity of the endpoint it takes as argument. We implement escape as the OCaml function

```
let escape u = (fun _ → let v, _ = create () in v) @> u
```

where the idea is to ignore the argument u and return one endpoint v from a freshly generated session. The lack of static checks concerning the linear usage of session endpoints makes this function well typed for OCaml. However, if we try to actually *use* escape, for example in the code

```
let _ =
  let u, _ = create () in close (escape u)
```

the runtime check in the implementation of @> detects the mismatch between u and v therefore causing the Error exception. ■

5.3 A Statically Checked Implementation

The second implementation we present reflects more accurately the typing information in endpoint types, which includes the identity of endpoints. In this case, we represent an endpoint type $[T]_\varrho$ as an OCaml type $(\tau_1,\tau_2,\varrho,\overline{\varrho})$ t where τ_1 and τ_2 are determined from T in a similar way as before. In addition, the phantom type parameter ϱ is the (abstract) identity of the endpoint and $\overline{\varrho}$ that of its peer (we represent identity variables as OCaml type variables and assume that $\overline{\varrho}$ is another OCaml type variable distinct from ϱ). More formally, the revised encoding of (session) types into OCaml types is given below:

Definition 5.4 (revised encoding of types and session types). Let $[\![\cdot]\!]_\iota$ and $\langle\!\langle\cdot\rangle\!\rangle$ be the encoding functions coinductively and mutually defined by the following equations:

$$
\begin{aligned}
[\![\mathbf{0}]\!]_\iota &= (\mathbf{0},\mathbf{0},\iota,\bar{\iota})\ \mathtt{t}\\
[\![\mathbf{1}]\!]_\iota &= (\mathbf{1},\mathbf{1},\iota,\bar{\iota})\ \mathtt{t}\\
[\![T;S]\!]_\iota &= ((\![T]\!]_\iota,[\![S]\!]_\iota)\ \mathtt{seq},([\![\overline{T}]\!]_{\bar{\iota}},[\![\overline{S}]\!]_{\bar{\iota}})\ \mathtt{seq},\iota,\bar{\iota})\ \mathtt{t}\\
[\![?t]\!]_\iota &= (\langle\!\langle t\rangle\!\rangle\ \mathtt{msg},\mathbf{0},\iota,\bar{\iota})\ \mathtt{t}\\
[\![!t]\!]_\iota &= (\mathbf{0},\langle\!\langle t\rangle\!\rangle\ \mathtt{msg},\iota,\bar{\iota})\ \mathtt{t}\\
[\![\&[\mathtt{C}_i : T_i]_{i\in I}]\!]_\iota &= (\{\mathtt{C}_i\ \mathtt{of}\ [\![T_i]\!]_\iota\}_{i\in I}\ \mathtt{tag},\mathbf{0},\iota,\bar{\iota})\ \mathtt{t}\\
[\![\oplus[\mathtt{C}_i : T_i]_{i\in I}]\!]_\iota &= (\mathbf{0},\{\mathtt{C}_i\ \mathtt{of}\ [\![\overline{T_i}]\!]_{\bar{\iota}}\}_{i\in I}\ \mathtt{tag},\iota,\bar{\iota})\ \mathtt{t}
\end{aligned}
$$

$$
\langle\!\langle [T]_\iota \rangle\!\rangle = [\![T]\!]_\iota
$$

where $\langle\!\langle\cdot\rangle\!\rangle$ is extended homomorphically to all the remaining type constructors erasing kind annotations on arrows and existential and universal quantifiers.

In Definition 5.4, ι is always an identity (co-)variable for we apply the encoding to user types in which these variables are never instantiated. Once again, the relation between the encoding of a session type and that of its dual can be expressed in terms of type equality:

Theorem 5.5. *If* $[\![T]\!]_\iota = (\tau_1,\tau_2,\iota,\bar{\iota})$ t, *then* $[\![\overline{T}]\!]_{\bar{\iota}} = (\tau_2,\tau_1,\bar{\iota},\iota)$ t.

The concrete representation of $(\alpha,\beta,\iota,\bar{\iota})$ t is the same that we have given in Section 5.2. As an optimization, the pol field of that representation could be omitted since there it is only necessary to verify an endpoint equality condition which is statically guaranteed by the implementation we are discussing now.

The easiest way of representing an existential type in OCaml is by means of its built-in module system [21]. In our case, we have to make sure that create returns a packaged pair of peer endpoints, each with its own identity. The OCaml representation of this type can be given by the following module signature

```
module type Package = sig
  type i and j
  val unpack : unit → (α,β,i,j) t × (β,α,j,i) t
```

```
  end
```

which contains two abstract type declarations `i` and `j`, corresponding to the identities of the two endpoints, and a function `unpack` to retrieve the endpoints once the module with this signature has been opened. Concerning the implementation of `Package`, there are two technical issues we have to address, both related to the fact that there cannot be two different endpoints with the same identity. First, we have to make sure that each session has its own implementation of the `Package` module signature. To this aim, we take advantage of `OCaml`'s support for *first-class modules* [8, 38], allowing us to write a function (`create` in the specific case) that returns a module implementation. The second issue is that we cannot store the two endpoints directly in the module, for the types of the endpoints contain type variables (α and β in the above signature) which are not allowed to occur free in a module. For this reason, we delay the actual creation of the endpoints at the time `unpack` is applied. This means, however, that the same implementation of `Package` could in principle be unpacked several times, instantiating different sessions whose endpoints would share the same identities. To make sure that `unpack` is applied at most once for each implementation of `Package` we resort once again to an atomic boolean flag.

The signatures of the functions implementing the communication primitives are essentially those we have already seen in Section 5.2, except for the presence of identity variables ϱ and σ and the type of `create`, which now returns a packaged pair of endpoints:

```
val create   : unit → (module Package)
val _send    : φ → (0,φ msg,ϱ,σ) t → (1,1,ϱ,σ) t
val _receive : (φ msg,0,ϱ,σ) t → φ × (1,1,ϱ,σ) t
val select   : ((β,α,σ,ϱ) t → φ) → (0,φ tag,ϱ,σ) t → (α,β,ϱ,σ) t
val branch   : (φ tag,0,ϱ,σ) t → φ
val (@=)     : ((α,β,ϱ,σ) t → φ × (1,1,ϱ,σ) t) →
               (((α,β,ϱ,σ) t,(γ,δ,ϱ,σ) t) seq,
                ((β,α,σ,ϱ) t,(δ,γ,σ,ϱ) t) seq,ϱ,σ) t → φ × (γ,δ,ϱ,σ) t
```

Note in particular the type of `select`, where we refer to both an endpoint and its peer by flipping the type parameters corresponding to session types (α and β) and those corresponding to identity variables (ϱ and σ) as well.

The implementation of `create` is shown below, in which `Previous.create` refers to the version of `create` detailed in Section 5.2:

```
let create () =
  let once = Flag.create () in
  (module struct
     type i and j
     let unpack () = Flag.use once; Previous.create ()
   end : Package)
```

The implementation of the I/O primitives is the same as in Section 5.2 and need not be repeated here. The resumption combinator shrinks to a simple cast

```
let (@=) = Obj.magic
```

since the equality condition on endpoints that is necessary for its soundness is now statically guaranteed by the type system. The cast is necessary because `@=` coerces its first argument to a function with a different type. With this implementation of `FuSe`$^{\{\}}$, a session is typically created thus

```
let module A = (val create ()) in (* create session   *)
let a, b = A.unpack () in         (* unpack endpoints *)
fork server a;                    (* fork server      *)
client b                          (* run client       *)
```

where `client` and `server` are suitable functions that use the two endpoints of the session without making any assumption on their identities. Otherwise, the abstract types `A.i` and `A.j` would escape their scope, resulting in a type error.

Example 5.6. A tentative version of the `escape` function discussed in Example 5.3 for the second `OCaml` implementation of `FuSe` is shown below:

```
let escape u = (fun _ → let module A = (val create ()) in
                        let v, _ = A.unpack () in v) @> u
```

`OCaml` correctly flags this version of `escape` as ill typed and emits an error message signalling the fact that the `i` type constructor in the locally opened module `A` would escape its scope. ■

Example 5.7. The function `bad_cast` of Example 4.11 can be implemented in `OCaml` as

```
let bad_cast x y = (fun x → swap ((fun y → (y, x)) @= y)) @= x
```

where `swap : α × β → β × α` permutes the two components of its pair argument. As we have seen in Example 4.11, `bad_cast` is well typed provided that its two parameters `x` and `y` are endpoints with the same identity. The code

```
let _ =
  let module A = (val create ()) in
  let u, v = A.unpack () in
  let u, v = bad_cast u v in
  close u; close v
```

attempts to use `bad_cast` to swap the types of `u` and `v`. `OCaml` correctly flags this code as ill typed since type `A.i` (the abstract type representing `u`'s identity) is not compatible with type `A.j` (the abstract type representing `v`'s identity). ■

6 EXTENDED EXAMPLE: TREES OVER SESSIONS

In this section we revisit and expand the running example of [31] to show how context-free session types help improve the precision of (inferred) protocols and the robustness of code. We start from the declaration

```
type α tree = Leaf | Node of α × α tree × α tree
```

defining an algebraic representation of binary trees, and we consider the following function, which sends a binary tree over a session endpoint. Note that, for the sake of readability, in this section we assume that `OCaml` polymorphic variant tags are curried as in the formal model and write for example `` `Node `` instead of its η-expansion `fun x → `Node x`.

```
1  let send_tree t u =
2    let rec send_tree_aux t u =
3      match t with
4      | Leaf → select `Leaf u
5      | Node (x, l, r) → let u = select `Node u in
6                         let u = send x u in
```

```
7                           let u = send_tree_aux l u in
8                           let u = send_tree_aux r u in u
9     in select `Done (send_tree_aux t u)
```

The auxiliary function `send_tree_aux` serializes a (sub)tree `t` on the endpoint `u`, whereas `send_tree` invokes `send_tree_aux` once and finally sends a sentinel label `Done` that signals the end of the stream of messages. FuSe infers for `send_tree` the type $\alpha\ \texttt{tree} \to T_{\texttt{reg}} \to A$ where $T_{\texttt{reg}}$ is the session type

$$T_{\texttt{reg}} = \oplus[\texttt{Leaf} : T_{\texttt{reg}}, \texttt{Node} : !\alpha; T_{\texttt{reg}}, \texttt{Done} : A] \tag{5}$$

and A is a session type variable (the code in `send_tree` does not specify in any way how `u` will be used when `send_tree` returns). Without the sentinel `Done`, the protocol $T_{\texttt{reg}}$ inferred by `OCaml` would never terminate (like $S_{\texttt{reg}}$ in (1)) making it hardly useful. Even with the sentinel, though, $T_{\texttt{reg}}$ is very imprecise. For example, it allows the labels `Node`, `Leaf`, and `Done` to be selected in this order, even though `send_tree` never generates such a sequence.[1]

To illustrate the sort of issues that this lack of precision may cause, it helps to look at a consumer process that receives a tree sent with `send_tree`:

```
1   let receive_tree u =
2     let rec receive_tree_aux u =
3       match branch u with
4       | `Leaf u → Leaf, u
5       | `Node u → let x, u = receive u in
6                   let l, u = receive_tree_aux u in
7                   let r, u = receive_tree_aux u in
8                   Node (x, l, r), u
9       | _ → assert false (* impossible *)
10    in let t, u = receive_tree_aux u in
11       match branch u with
12       | `Done u → (t, u)
13       | _ → assert false (* impossible *)
```

This function consists of a main body (lines 2–9) responsible for building up a (sub)tree received from `u`, the bootstrap of the reception phase (line 10), and a final reception that awaits for the sentinel (lines 11–13). For `receive_tree`, `OCaml` infers the type $\overline{T_{\texttt{reg}}} \to \alpha\ \texttt{tree} \times \overline{A}$. The fact that `send_tree` and `receive_tree` use endpoints with dual session types should be enough to reassure us that the two functions communicate safely within the same session. Unfortunately, our confidence is spoiled by two suspicious catch-all cases (lines 9 and 13) without which `receive_tree` would be ill typed. In particular, omitting line 9 would result in a non-exhaustive pattern matching (lines 3–8) because label `Done` can in principle be received along with `Leaf` and `Node`. A similar issue would arise omitting line 13. Omitting both lines 9 and 13 would also be a problem. In search of a typing derivation for `receive_tree`, `OCaml` would try to compute the intersection of the labels handled by the two pattern matching constructs, only to find out that such intersection is empty.

We clean up and simplify `send_tree` and `receive_tree` using resumptions:

```
1   let rec send_tree t u =
2     match t with
```

[1] The claim made in [31] that `send_tree_aux` is ill typed is incorrect. There exist typing derivations for `send_tree_aux` proving that it has type $\alpha\ \texttt{tree} \to T \to T$ for every T that satisfies the equation $T = \oplus[\texttt{Leaf} : T, \texttt{Node} : !\alpha; T, \ldots]$.

```
3     | Leaf → select `Leaf u
4     | Node (x, l, r) → let u = select `Node u in
5                         let u = send x u in
6                         let u = send_tree l @> u in (* resumption *)
7                         let u = send_tree r u in u
8   let rec receive_tree u =
9     match branch u with
10    | `Leaf u → Leaf, u
11    | `Node u → let x, u = receive u in
12                let l, u = receive_tree @= u in    (* resumption *)
13                let r, u = receive_tree u in Node (x, l, r), u
```

In `send_tree` we use the simple resumption `@>` since the function only returns the endpoint `u`. In `receive_tree` we use `@=` since the function returns the received tree in addition to the continuation endpoint. We no longer need an explicit sentinel message `Done` that marks the end of the message stream because the protocol now specifies exactly the number of messages needed to serialize a tree. For the same reason, the catch-all cases in `receive_tree` are no longer necessary. For these functions, `OCaml` respectively infers the types $\alpha\ \texttt{tree} \to [T_{\texttt{cf}}]_{\varrho} \to [\mathbf{1}]_{\varrho}$ and $[\overline{T_{\texttt{cf}}}]_{\varrho} \to \alpha\ \texttt{tree} \times [\mathbf{1}]_{\varrho}$ where $T_{\texttt{cf}}$ is the session type such that

$$T_{\texttt{cf}} = \oplus[\texttt{Leaf} : \mathbf{1}, \texttt{Node} : !\alpha; T_{\texttt{cf}}; T_{\texttt{cf}}]$$

The leftmost occurrence of $\_;\_$ in $!\alpha; T_{\texttt{cf}}; T_{\texttt{cf}}$ is due to the communication primitive (either `_send` or `_receive`) and the rightmost one to the resumption.

Note that the only difference between the revised `send_tree` and the homonymous function presented in [31] is the occurrence of `@>`. All the other examples in [31] can be patched similarly by resuming endpoints at the appropriate places.

7 RELATED WORK

The work most closely related to ours is [31] in which Thiemann and Vasconcelos introduce context-free session types, develop their metatheory, and prove that session type equivalence is decidable. The main difference between Thiemann and Vasconcelos' type system and our own concerns the treatment of sequential compositions in session types. In [31], the only typing rules that can eliminate a sequential composition $T; S$ are those concerning `send` and `receive`. As a consequence, all the interesting typing derivations use the monoidal laws of sequential composition in a fundamental way: when typing an occurrence of `send` or `receive`, sequential compositions should be associated so that T is the single input/output operation being performed; when typing a (recursive) function application, sequential compositions should be associated so that T describes the whole prefix of the protocol carried out by the function and S the residual protocol. The different ways sequential compositions must be associated in different parts of the code are dealt with a type system featuring (1) a structural rule that rearranges sequential compositions in session types and (2) support for polymorphic recursion. In contrast, our approach relies on the use of resumptions inserted in the code. Resumptions tie sequential compositions in session types to the structure of the code that uses endpoints with those types. As a consequence, equivalent session types (in the sense of Definition 3.2) are no longer interchangeable, but the resulting type system does not need structural rules nor polymorphic recursion and so it is more amenable to be embedded in a host programming language taking advantage of its type inference engine.

Overall, we think that our approach strikes a good balance between expressiveness and flexibility: resumptions are unobtrusive and typically sparse and they give the programmer complete control

over the occurrences of sequential compositions in session types, resolving the ambiguities that arise with context-free session type inference (Sections 1 and 6). Also, the locations where resumptions should be used in the code are typically easy to spot thanks to a simple but effective rule of thumb: place resumptions in recursive applications that are *not* in tail position. The rationale for this rule is simple: the recursive call carries out some prefix of a communication protocol; the fact that the call is not in tail position suggests that after the call some further suffix of the protocol also needs to be carried out. All the examples we have discussed, including the ones in [31], can be addressed applying this rule. Finally, the resumption-based handling of sequential compositions allows subtyping to some extent, despite the fact that subtyping for context-free session types in undecidable in general (Proposition 3.4). Indeed, as explained in the implementation of `FuSe` [26], the type parameters of endpoint types can be decorated with variance annotations allowing `OCaml`'s subtyping to be lifted at the level of session types. This reduced form of subtyping does not account for the monoidal and distributive laws of sequential composition, but it includes the well-known co-/contra-variance properties of branches and choices [9] which happen to be useful in practice.

A potential limitation of our approach compared to [31] is that we require processes operating on peer endpoints of a session to mirror each other as far as the placement of resumptions is concerned. For example, a process using an endpoint with type $(!\texttt{int};\mathbf{1});?\texttt{bool}$ may interact with another process that uses an endpoint with type $(?\texttt{int};\mathbf{1});!\texttt{bool}$, but not with a process using an endpoint with type $?\texttt{int};!\texttt{bool}$ even though $(?\texttt{int};\mathbf{1});!\texttt{bool} \sim ?\texttt{int};!\texttt{bool}$. Both processes must resume the endpoints they use after the exchange of the first message. Understanding the practical impact of this limitation requires an extensive analysis of code that deals with context-free protocols. We have not pursued such investigation, but we can make two observations nonetheless. First, resumptions are often used in combination with recursion and interacting recursive processes already tend to mirror each other by their own recursive nature. We can see this by comparing `send_tree` and `receive_tree` (Section 6) and also by looking at the examples in [31]. Second, it is easy to provide *explicit coercions* corresponding to laws of $\sim$. Such coercions, whose soundness is already accounted for by Theorem 4.5, can be used to rearrange sequential compositions in session types. For example, a coercion $(A;\mathbf{1});B \to A;B$ composed with a function $?\texttt{int};!\texttt{bool} \to \alpha$ would turn it into a function $(?\texttt{int};\mathbf{1});!\texttt{bool} \to \alpha$. The use of coercions augments the direct involvement of the programmer, but is a low-cost solution to broaden the cases already addressed by plain resumptions. One natural question to ask is whether the type error raised by `OCaml` due to a missing coercion would provide any hint as to where the coercion should be placed. Unfortunately, the answer is often negative in this respect. The point is that a session type error due to a mismatch between the protocols followed by client and server is usually flagged at the point in the code where client and server are connected together. This may be very far away from the point (in the body of either client or server) where the coercion may be necessary. This is a notable case where native session type support from the compiler may help producing meaningful error messages, something that a library implementation falls short of.

`FuSe` [26] is an `OCaml` implementation of binary sessions that combines static protocol enforcement with runtime checks for endpoint linearity [13, 33] and resumption safety (Section 5.2). Support for sequential composition of session types based on resumptions was originally introduced in `FuSe` to describe *iterative protocols*, showing that a class of unbounded protocols could be described without resorting to (equi-)recursive types. The work of Thiemann and Vasconcelos [31] prompted us to formalize resumptions and to study their implications for the precision of protocol descriptions. This led to the discovery of a bug in early versions of `FuSe` where peer endpoints were given the same identity (*cf.* the discussion at the end of Example 4.11) and then to the development of a fully static typing discipline to enforce resumption safety (Sections 3 and 5.3).

The need to express protocols going beyond regular expressions has been recognized also by Igarashi and Kobayashi [15], who devise a type system for analyzing the usage protocols of resources such as memory regions, locks and files. Interestingly, their type system makes no use of polymorphic recursion (unlike Thiemann and Vasconcelos [31]) nor of specific operators akin to resumptions (unlike our approach). The type system is also generic, in the sense that it enables reasoning on arbitrary resources, each characterized by a specific set of operations. Igarashi and Kobayashi [15] present an inference algorithm for their type system. In contrast, our approach addresses the problem of providing an off-the-shelf solution to context-free session type inference using a general-purpose programming language.

The use of type variables abstracting over the identity of endpoints has been inspired by works on regions and linear types [2, 36], by L^3 [1], a language with locations supporting strong updates, and Alms [32, 34], an experimental general-purpose programming language with affine types. In these works, abstract identities are used to associate an object with the region it belongs to [2, 36], or to link the (non-linear) reference to a mutable object with the (linear or affine) capability for accessing it. Interestingly, in these works separating the reference from the capability (hence the use of abstract identities) is not really a necessity, but rather a technique that results in increased flexibility: the reference can be aliased without restrictions to create cyclic graphs [1] or to support "dirty" operations on shared data structures [34]. In our case, endpoint identities are crucial for checking the safety of resumptions. As one of the anonymous reviewers pointed out, the technique of using type variables abstracting over regions can be traced back to the implementation of stateful computations in Haskell [20], which was elaborated and proven safe by Moggi and Sabry [22].

ACKNOWLEDGMENTS

The author is grateful to the anonymous reviewers whose comments and questions have led to substantial improvements in both content and presentation of the article. The author has also benefited from discussions with, and references provided by, Naoki Kobayashi, Peter Thiemann and Vasco T. Vasconcelos.

REFERENCES

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundamenta Informaticae* 77, 4 (2007), 397–449.

[2] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP'08)*. ACM, 213–224. DOI: https://doi.org/10.1145/1411204.1411235

[3] Bruno Courcelle. 1983. Fundamental Properties of Infinite Trees. *Theoretical Computer Science* 25 (1983), 95–169. DOI: https://doi.org/10.1016/0304-3975(83)90059-2

[4] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Information and Computation* 256 (2017), 253–286. DOI: https://doi.org/10.1016/j.ic.2017.06.002

[5] Romain Demangeon and Kohei Honda. 2011. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR'11) (LNCS)*, Vol. 6901. Springer, 280–296. DOI: https://doi.org/10.1007/978-3-642-23217-6_19

[6] Gert Florijn. 1995. Object Protocols as Functional Parsers. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95) (LNCS)*, Vol. 952. Springer, 351–373. DOI: https://doi.org/10.1007/3-540-49538-X_17

[7] Emily P. Friedman. 1976. The Inclusion Problem for Simple Languages. *Theoretical Computer Science* 1, 4 (1976), 297–316. DOI: https://doi.org/10.1016/0304-3975(76)90074-8

[8] Alain Frisch and Jacques Garrigue. 2010. First-class modules and composable signatures in Objective Caml 3.12. In *ACM SIGPLAN Workshop on ML*.

[9] Simon Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-calculus. *Acta Informatica* 42, 2-3 (2005), 191–225. DOI: https://doi.org/10.1007/s00236-005-0177-z

[10] Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50. DOI: https://doi.org/10.1017/S0956796809990268

[11] Kohei Honda. 1993. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93) (LNCS)*, Vol. 715. Springer, 509–523. DOI: https://doi.org/10.1007/3-540-57208-2_35

[12] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming (ESOP'98) (LNCS)*, Vol. 1381. Springer, 122–138. DOI: https://doi.org/10.1007/BFb0053567

[13] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification through Endpoint API Generation. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE'16) (LNCS)*, Vol. 9633. Springer, 401–418. DOI: https://doi.org/10.1007/978-3-662-49665-7_24

[14] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *Comput. Surveys* 49, 1 (2016), 3. DOI: https://doi.org/10.1145/2873052

[15] Atsushi Igarashi and Naoki Kobayashi. 2005. Resource Usage Analysis. *ACM Transactions on Programming Languages and Systems* 27, 2 (March 2005), 264–313. DOI: https://doi.org/10.1145/1057387.1057390

[16] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. 1993. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems* 15, 2 (1993), 290–311. DOI: https://doi.org/10.1145/169701.169687

[17] Naoki Kobayashi. 2002. Type Systems for Concurrent Programs. In *10th Anniversary Colloquium of UNU/IIST (LNCS)*, Vol. 2757. Springer, 439–453. DOI: https://doi.org/10.1007/978-3-540-40007-3_26 Extended version available at http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf.

[18] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems* 21, 5 (1999), 914–947. DOI: https://doi.org/10.1145/330249.330251

[19] A. J. Korenjak and John E. Hopcroft. 1966. Simple Deterministic Languages. In *Proceedings of 7th Annual Symposium on Switching and Automata Theory (SWAT'66)*. IEEE Computer Society, 36–46. DOI: https://doi.org/10.1109/SWAT.1966.22

[20] John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *Lisp and Symbolic Computation* 8, 4 (1995), 293–341.

[21] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems* 10, 3 (1988), 470–502. DOI: https://doi.org/10.1145/44501.45065

[22] Eugenio Moggi and Amr Sabry. 2001. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming* 11, 6 (2001), 591–627. DOI: https://doi.org/10.1017/S0956796801004154

[23] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *Proceedings of the 18th European Symposium on Programming (ESOP'09) (LNCS)*, Vol. 5502. Springer, 316–332. DOI: https://doi.org/10.1007/978-3-642-00590-9_23

[24] Oscar Nierstrasz. 1993. Regular Types for Active Objects. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*. ACM, 1–15. DOI: https://doi.org/10.1145/165854.167976

[25] Luca Padovani. 2016. FuSe – A Simple Library Implementation of Binary Sessions. (2016). Retrieved May 27, 2018 from http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html

[26] Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming* 27 (2017). DOI: https://doi.org/10.1017/S0956796816000289

[27] Luca Padovani. 2017. Context-Free Session Type Inference. In *Proceedings of the 26th European Symposium on Programming (ESOP'17) (LNCS)*, Vol. 10201. Springer, 804–830. DOI: https://doi.org/10.1007/978-3-662-54434-1_30

[28] António Ravara and Vasco T. Vasconcelos. 2000. Typing Non-uniform Concurrent Objects. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00) (LNCS)*, Vol. 1877. Springer, 474–488. DOI: https://doi.org/10.1007/3-540-44618-4_34

[29] John H. Reppy. 1999. *Concurrent Programming in ML*. Cambridge University Press.

[30] Mario Südholt. 2005. A Model of Components with Non-regular Protocols. In *Revised Selected Papers of the 4th International Workshop on Software Composition (SC'05) (LNCS)*, Vol. 3628. Springer, 99–113. DOI: https://doi.org/10.1007/11550679_8

[31] Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-Free Session Types. In *Proceedings of the 21st International Conference on Functional Programming (ICFP'16)*. ACM, 462–475. DOI: https://doi.org/10.1145/2951913.2951926

[32] Jesse A. Tov. 2012. *Practical Programming with Substructural Types*. Ph.D. Dissertation. Northeastern University.

[33] Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Proceedings of the 19th European Symposium on Programming (ESOP'10) (LNCS)*, Vol. 6012. Springer, 550–569. DOI: https://doi.org/10.1007/978-3-642-11957-6_29

[34] Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages (POPL'11)*. ACM, 447–458. DOI: https://doi.org/10.1145/1926385.1926436

[35] Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418. DOI: https://doi.org/10.1017/S095679681400001X
[36] David Walker and Kevin Watkins. 2001. On Regions and Linear Types. In *Proceedings of the 6th International Conference on Functional Programming (ICFP'01)*. ACM, 181–192. DOI: https://doi.org/10.1145/507635.507658
[37] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. DOI: https://doi.org/10.1006/inco.1994.1093
[38] Jeremy Yallop and Oleg Kiselyov. 2010. First-class modules: hidden power and tantalizing promises. In *ACM SIGPLAN Workshop on ML*.

A PROPERTIES OF SUBTYPING

Proposition 3.3. *The following properties hold:*

(1) *(associativity)* $T;(S;R) \sim (T;S);R$;
(2) *(unit)* $\mathbf{1};T \sim T;\mathbf{1} \sim T$.
(3) *(distributivity)* $\&[\mathtt{C}_i : T_i]_{i\in I};T \sim \&[\mathtt{C}_i : T_i;T]$ *and* $\oplus[\mathtt{C}_i : T_i]_{i\in I};T \sim \oplus[\mathtt{C}_i : T_i;T]$;
(4) *(pre-congruence)* $T \lesssim T'$ *and* $S \lesssim S'$ *imply* $T;S \lesssim T';S'$.

Proof. All properties are proved by simple applications of the coinduction principle. For illustrative purposes we focus on item (4). It suffices to show that

$$\mathcal{R} \stackrel{\text{def}}{=} \lesssim \cup \{(T_1;T_2, S_1;S_2) \mid T_1 \lesssim S_1 \wedge T_2 \lesssim S_2\}$$

is included in $\lesssim$. Suppose $(T_1;T_2, S_1;S_2) \in \mathcal{R}$ where $T_i \lesssim S_i$ for $i = 1, 2$. We reason by cases on the conditions 1–4 of Definition 3.2 concerning T_i and S_i. We omit the cases for output actions which are analogous to those for input actions:

- Suppose terminated(T_1). From condition 1 of Definition 3.2 we deduce terminated(S_1). By definition of terminated($\cdot$) we conclude terminated($T_1;T_2$) and terminated($S_1;S_2$), hence condition 1 of Definition 3.2 is satisfied.
- Suppose done(T_1) and done(T_2). From condition 2 of Definition 3.2 we deduce done(S_1) and done(S_2). By definition of done($\cdot$) we conclude done($T_1;T_2$) and done($S_1;S_2$), hence condition 2 of Definition 3.2 is satisfied.
- Suppose done(T_1) and $T_2 \xrightarrow{?\omega} T_2'$. From condition 3 of Definition 3.2 we deduce done(S_1) and $S_2 \xrightarrow{?\omega} S_2'$ and $T_2' \lesssim S_2'$. We conclude that condition 3 of Definition 3.2 is satisfied by observing that $T_1;T_2 \xrightarrow{?\omega} T_2'$ and $S_1;S_2 \xrightarrow{?\omega} S_2'$ and by definition of $\mathcal{R}$.
- Suppose $T_1 \xrightarrow{?\omega} T_1'$. From condition 3 of Definition 3.2 we deduce $S_1 \xrightarrow{?\omega} S_1'$ and $T_1' \lesssim S_1'$. We conclude that condition 3 of Definition 3.2 is satisfied by observing that $T_1;T_2 \xrightarrow{?\omega} T_1';T_2$ and $S_1;S_2 \xrightarrow{?\omega} S_1';S_2$ and by definition of $\mathcal{R}$. □

Next we have a standard result relating subtyping and duality. The result has an important corollary because it guarantees that duality preserves equivalence, that is $T \sim S$ implies $\overline{T} \sim \overline{S}$.

Proposition A.1. *If* $T \lesssim S$, *then* $\overline{S} \lesssim \overline{T}$.

Proof. It suffices to show that $\mathcal{R} \stackrel{\text{def}}{=} \{(\overline{S}, \overline{T}) \mid T \lesssim S\}$ satisfies the conditions of Definition 3.2. The details are easy and omitted. □

Proposition 3.4. *The relation* $\sim$ *is decidable, whereas* $\lesssim$ *is not.*

Proof. Decidability of $\sim$ has already been shown in [31]. We prove the undecidability of $\lesssim$ by reducing the inclusion problem for simple deterministic languages [19] to subtyping for context-free session types. The inclusion problem for simple deterministic languages has been shown

undecidable by Friedman [7]. Following the terminology of Korenjak and Hopcroft [19], we say that a context-free grammar $\mathcal{G} = (\mathsf{V}, \mathsf{T}, \mathsf{S}, \mathsf{P})$ is an *s-grammar* if:

- every production in P is of the form $Z \to aY_1 \cdots Y_n$ where $n \geq 0$, $Z \in \mathsf{V}$ is called *head* of the production, $Y_i \in \mathsf{V}$ for all $1 \leq i \leq n$, and $a \in \mathsf{T}$ is called *handle* of the production;
- there are no two distinct productions in P with the same head Z and the same handle a.

The basic idea of the proof is to associate each s-grammar $\mathcal{G}$ with a session type $\mathrm{st}(\mathcal{G})$ such that the language $L(\mathcal{G})$ generated by $\mathcal{G}$ coincides with the complete traces of $\mathrm{st}(\mathcal{G})$. Then, the inclusion problem between two languages respectively generated by the grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ corresponds to the relation $\mathrm{st}(\mathcal{G}_1) \lesssim \mathrm{st}(\mathcal{G}_2)$.

To define $\mathrm{st}(\mathcal{G})$ from $\mathcal{G} = (\mathsf{V}, \mathsf{T}, \mathsf{S}, \mathsf{P})$ we assume, without loss of generality, that $\mathsf{V} = \{X_1, \ldots, X_n\}$, $\mathsf{T} = \{\mathsf{C}_1, \ldots, \mathsf{C}_m\}$ and $\mathsf{S} = X_1$. From $\mathcal{G}$ we define the (finite) system of equations

$$X_i = \&[\mathsf{C} : Y_1 ; \cdots ; Y_k]_{X_i \to \mathsf{C} Y_1 \cdots Y_k \in \mathsf{P}} \qquad (1 \leq i \leq n)$$

where the sequential composition $Y_1 ; \cdots ; Y_k$ is $\mathbf{1}$ when $k = 0$. We appeal to Courcelle [3] to deduce that this system of equations has a unique solution $\{X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\}$ and we identify $\mathrm{st}(\mathcal{G})$ with T_1, that is the component of the solution corresponding to the start symbol of $\mathcal{G}$.

Now it is a simple exercise to verify that

$$\mathrm{st}(\mathcal{G}) \xrightarrow{?\mathsf{C}_1} \cdots \xrightarrow{?\mathsf{C}_k} \mathbf{1} \iff \mathsf{C}_1 \cdots \mathsf{C}_k \in L(\mathcal{G})$$

and therefore that

$$L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2) \iff \mathrm{st}(\mathcal{G}_1) \lesssim \mathrm{st}(\mathcal{G}_2)$$

since external choices are covariant with respect to subtyping (Definition 3.2). This is enough to conclude that $\lesssim$ is undecidable. □

B SOUNDNESS OF THE TYPE SYSTEM

B.1 Auxiliary notions for type environments

Note that every environment Γ such that $\Gamma :: \mathsf{U}$ is trivially balanced, there are no endpoints in its domain.

Proposition B.1. *Let $\Gamma \overset{\mu}{\longmapsto} \Gamma'$. The following properties hold:*

(1) *If Γ is ground/well formed/balanced, then so is Γ'.*

(2) *If $\Gamma + \Gamma''$ is defined, then so is $\Gamma' + \Gamma''$ and $\Gamma + \Gamma'' \overset{\mu}{\longmapsto} \Gamma' + \Gamma''$.*

Proof. Trivial from Definitions 4.3 and 4.4. □

B.2 Subject reduction for expressions

Definition B.2 (subkinding). Let $\leq$ be the least total order on kinds such that $\mathsf{U} \leq \mathsf{L}$. We say that κ_1 is *subkind* of κ_2 if $\kappa_1 \leq \kappa_2$.

The following lemma relates the kind of a value with that of the environment in which it is typed.

Lemma B.3. *If $\Delta; \Gamma \vdash v : t$ and $t :: \kappa$, then $\Gamma :: \kappa$.*

Proof. By induction on the structure of v and by cases on its shape.

Case $v = \mathsf{c}\, v_1 \cdots v_n$ where $\mathsf{c} \in \{(), \mathtt{pair}, \mathsf{C}, \mathtt{fix}, \mathtt{fork}, \mathtt{\_send}, \mathtt{select}\}$. From [T-APP] and [T-CONST] we deduce $\Gamma = \Gamma_0 + \Gamma_1 + \cdots + \Gamma_n$ and $\Gamma_0 :: \mathsf{U}$ and $\Delta; \Gamma_0 \vdash \mathsf{c} : t_1 \to^{\kappa_1} \cdots \to^{\kappa_{n-1}} t_n \to^{\kappa_n} t$ and $\Delta; \Gamma_i \vdash v_i : t_i$ for every $1 \leq i \leq n$. By inspecting Table 3 we observe that $t :: \kappa$ implies $t_i :: \kappa$ for every $1 \leq i \leq n$. By induction hypothesis we deduce $\Gamma_i :: \kappa$ for every $1 \leq i \leq n$, hence we conclude $\Gamma :: \kappa$.

Case $v = \lambda x.e$. From [T-FUN] we deduce $t = t_1 \to^{\kappa'} t_2$ where $\kappa' \leq \kappa$ and $\Gamma :: \kappa'$, hence we conclude $\Gamma :: \kappa$.

Case $v = \Lambda\varrho.w$. From [T-ID-FUN] we deduce $t = \forall\varrho.s$ and $\Delta, \varrho; \Gamma \vdash w : s$. From the hypothesis $t :: \kappa$ and Definition 3.5 we deduce $s :: \kappa$. By induction hypothesis we conclude $\Gamma :: \kappa$.

Case $v = \varepsilon$. Then $\Gamma = \Gamma', \varepsilon : t$. From [T-NAME] we deduce $\Gamma' :: \mathsf{U}$ hence we conclude $\Gamma :: \kappa$.

Case $v = \lceil \varepsilon, w \rceil$. From [T-PACK] we deduce $t = \exists\varrho.s$ and $\Delta; \Gamma \vdash w : s\{\varepsilon/\varrho\}$. From the hypothesis $t :: \kappa$ and Definition 3.5 we deduce $s :: \kappa$ hence $s\{\varepsilon/\varrho\} :: \kappa$. By induction hypothesis we conclude $\Gamma :: \kappa$. □

Hereafter we write $\mathsf{fn}(e)$ for the set of names occurring free in e.

Lemma B.4. *If* $\Delta; \Gamma \vdash e : t$, *then* $\mathsf{fn}(e) \subseteq \mathsf{dom}(\Gamma)$.

Proof. A simple induction on the derivation of $\Delta; \Gamma \vdash e : t$. □

The type system enjoys an almost standard weakening property. Note that weakening is allowed for unlimited environments only.

Lemma B.5 (weakening). *If* $\Delta; \Gamma_1 \vdash e : t$ *and* $\Gamma_1 + \Gamma_2$ *is defined and* $\Gamma_2 :: \mathsf{U}$, *then* $\Delta; \Gamma_1 + \Gamma_2 \vdash e : t$.

Proof. A simple induction on the derivation of $\Delta; \Gamma_1 \vdash e : t$. □

Lemma B.6 (value substitution). *If* (1) $\Delta; \Gamma_1, x : s \vdash e : t$ *and* (2) $\Delta; \Gamma_2 \vdash v : s$ *and* (3) $\Gamma_1 + \Gamma_2$ *is defined, then* $\Delta; \Gamma_1 + \Gamma_2 \vdash e\{v/x\} : t$.

Proof. By induction on the derivation of (1) and by cases on the last rule applied. We only discuss a few significant cases.

Case [T-CONST]. Then $e = \mathsf{c}$ for some constant c and $(\Gamma_1, x : s) :: \mathsf{U}$. In particular, $s :: \mathsf{U}$. From Lemma B.3 we deduce $\Gamma_2 :: \mathsf{U}$ and we conclude with Lemma B.5.

Case [T-NAME]. Then $e = u$ and $\Gamma_1, x : s = \Gamma', u : s'$ where $\Gamma' :: \mathsf{U}$. We distinguish three subcases, depending on whether u is an endpoint or a variable other than x or x itself. If $u = \varepsilon$ or $u = y \neq x$, then $s :: \mathsf{U}$. From Lemma B.3 we deduce $\Gamma_2 :: \mathsf{U}$ and we conclude with Lemma B.5. If $u = x$, then $\Gamma_1 = \Gamma'$ and $e\{v/x\} = v$ and we conclude from (2) and Lemma B.5.

Case [T-FUN]. Then $e = \lambda y.e'$. Without loss of generality we may assume that $x \neq y$, hence from [T-FUN] we deduce $t = t_1 \to^{\kappa} t_2$ and $\Delta; \Gamma_1, x : s, y : t_1 \vdash e' : t_2$ and $(\Gamma_1, x : s) :: \kappa$. From Lemma B.3 we deduce $\Gamma_2 :: \kappa$. By induction hypothesis we deduce $\Delta; (\Gamma_1 + \Gamma_2), y : t_1 \vdash e'\{v/x\} : t_2$ where $\Gamma_1 + \Gamma_2 :: \kappa$. We conclude with one application of [T-FUN].

Case [T-APP]. Then $e = e_1\ e_2$ and $\Gamma, x : s = \Gamma_{11} + \Gamma_{12}$ where $\Delta; \Gamma_{11} \vdash e_1 : t' \to^{\kappa} t$ and $\Delta; \Gamma_{12} \vdash e_2 : t'$. We distinguish three sub-cases:

- If $x \in \mathsf{dom}(\Gamma_{11}) \setminus \mathsf{dom}(\Gamma_{12})$, then from Lemma B.4 we deduce $x \notin \mathsf{fn}(e_2)$ and $\Gamma_{11} = \Gamma'_{11}, x : s$. By induction hypothesis we deduce $\Delta; \Gamma'_{11} + \Gamma_2 \vdash e_1\{v/x\} : t' \to^{\kappa} t$ and we conclude with one application of [T-APP].
- If $x \in \mathsf{dom}(\Gamma_{12}) \setminus \mathsf{dom}(\Gamma_{11})$, then from Lemma B.4 we deduce $x \notin \mathsf{fn}(e_1)$ and $\Gamma_{12} = \Gamma'_{12}, x : s$. By induction hypothesis we deduce $\Delta; \Gamma'_{12} + \Gamma_2 \vdash e_2\{v/x\} : t'$ and we conclude with one application of [T-APP].
- If $x \in \mathsf{dom}(\Gamma_{11}) \cap \mathsf{dom}(\Gamma_{12})$, then $\Gamma_{11} = \Gamma'_{11}, x : s$ and $\Gamma_{12} = \Gamma'_{12}, x : s$. From Definition 4.1 we deduce $s :: \mathsf{U}$ and, from Lemma B.3, also $\Gamma_2 :: \mathsf{U}$. Therefore, $\Gamma_2 + \Gamma_2 = \Gamma_2$, again by Definition 4.1. By induction hypothesis we derive $\Delta; \Gamma'_{11} + \Gamma_2 \vdash e_1\{v/x\} : t' \to^{\kappa} t$ and $\Delta; \Gamma'_{12} + \Gamma_2 \vdash e_2\{v/x\} : t'$. We conclude with one application of [T-APP]. □

Lemma B.7 (identity substitution). *If (1) $\Delta, \varrho; \Gamma \vdash e : t$ and (2) $\varepsilon \in \Delta$, then $\Delta; \Gamma\{\varepsilon/\varrho\} \vdash e\{\varepsilon/\varrho\} : t\{\varepsilon/\varrho\}$.*

Proof. A simple induction on the derivation of (1). □

Theorem 4.5 (subject reduction for expressions). *Let $\Delta; \Gamma \vdash e : t$ where Γ is ground and well formed. If $e \longrightarrow e'$, then $\Delta; \Gamma' \vdash e' : t$ for some $\Gamma' \sim \Gamma$.*

Proof. By cases on the rule used to derive $e \longrightarrow e'$. We only discuss a few representative cases.

Case [R1]. Then $e = (\lambda x.e'')\ v \longrightarrow e''\{v/x\} = e'$. From [T-APP] we deduce that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_1 \vdash \lambda x.e'' : s \to^\kappa t$ and $\Delta; \Gamma_2 \vdash v : s$. From [T-FUN] we deduce that $\Delta; \Gamma_1, x : s \vdash e'' : t$ and $\Gamma_1 :: \kappa$. Using Lemma B.6 we deduce $\Delta; \Gamma_1 + \Gamma_2 \vdash e''\{v/x\} : t$ and we conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma$.

Case [R2]. Then $e = (\Lambda\varrho.v)\ [\varepsilon] \longrightarrow v\{\varepsilon/\varrho\} = e'$. From [T-ID-APP] we deduce that $\Delta; \Gamma \vdash \Lambda\varrho.v : \forall\varrho.s$ and $\varepsilon \in \Delta$ and $t = s\{\varepsilon/\varrho\}$. From [T-ID-FUN] we deduce that $\Delta, \varrho; \Gamma \vdash v : s$. From Lemma B.7 we deduce $\Delta; \Gamma\{\varepsilon/\varrho\} \vdash v\{\varepsilon/\varrho\} : s\{\varepsilon/\varrho\}$. From the (implicit) hypothesis that every typing judgment is well formed we deduce $\varrho \notin \mathrm{fid}(\Gamma)$, hence $\Gamma\{\varepsilon/\varrho\} = \Gamma$. We conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma$.

Case [R3]. Then $e = \mathtt{fix}\ v\ w \longrightarrow v\ (\mathtt{fix}\ v)\ w = e'$. From [T-APP] and the type scheme of $\mathtt{fix}$ we deduce that $\Gamma = \Gamma_1 + \Gamma_2 + \Gamma_3$ and $\Delta; \Gamma_1 \vdash \mathtt{fix} : ((s \to t) \to s \to t) \to s \to t$ and $\Delta; \Gamma_2 \vdash v : (s \to t) \to s \to t$ and $\Delta; \Gamma_3 \vdash w : s$. From Lemma B.3 we deduce $\Gamma_1 :: \mathsf{U}$ and $\Gamma_2 :: \mathsf{U}$. In particular, $\Gamma_2 = \Gamma_2 + \Gamma_2$. From [T-APP] we deduce $\Delta; \Gamma_1 + \Gamma_2 \vdash v\ (\mathtt{fix}\ v) : s \to t$. From [T-APP] we deduce $\Delta; \Gamma \vdash e' : t$ and we conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma$.

Case [R6]. Then $e = \mathtt{let}\ \lceil \varrho, x \rceil = \lceil \varepsilon, v \rceil\ \mathtt{in}\ e'' \longrightarrow e''\{\varepsilon/\varrho\}\{v/x\} = e'$. From [T-UNPACK] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_1 \vdash \lceil \varepsilon, v \rceil : \exists\varrho.s$ and $\Delta, \varrho; \Gamma_2, x : s \vdash e'' : t$. From [T-PACK] we deduce $\Delta; \Gamma_1 \vdash v : s\{\varepsilon/\varrho\}$ and $\varepsilon \in \Delta$. From the (implicit) assumption that every typing judgment is well formed we deduce $\varrho \notin \mathrm{fid}(\Gamma_2) \cup \mathrm{fid}(t)$, hence $\Gamma_2\{\varepsilon/\varrho\} = \Gamma_2$ and $t\{\varepsilon/\varrho\} = t$. From Lemma B.7 we deduce $\Delta; \Gamma_2, x : s\{\varepsilon/\varrho\} \vdash e''\{\varepsilon/\varrho\} : t$. From Lemma B.6 we deduce $\Delta; \Gamma_1 + \Gamma_2 \vdash e''\{\varepsilon/\varrho\}\{v/x\} : t$ and we conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma$.

Case [R7]. Then $e = \{(v, \varepsilon)\}_\varepsilon \longrightarrow (v, \varepsilon) = e'$. From [T-RESUME] and the hypothesis that Γ is well formed we deduce $\Gamma = \Gamma'', \varepsilon : [T; S]_\varepsilon$ and $\Delta; \Gamma'', \varepsilon : [T]_\varepsilon \vdash (v, \varepsilon) : s \times [\mathbf{1}]_\varepsilon$ and $t = s \times [S]_\varepsilon$. From [T-APP] and the type scheme of $\mathtt{pair}$ we deduce that $\Gamma'', \varepsilon : [T]_\varepsilon = \Gamma_0 + \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_0 \vdash \mathtt{pair} : s \to [\mathbf{1}]_\varepsilon \to^\kappa s \times [\mathbf{1}]_\varepsilon$ and $\Delta; \Gamma_1 \vdash v : s$ and $\Delta; \Gamma_2 \vdash \varepsilon : [\mathbf{1}]_\varepsilon$ where $\Gamma_0 :: \mathsf{U}$ and $s :: \kappa$. From [T-NAME] we deduce that $T = \mathbf{1}$. We conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma'', \varepsilon : [S]_\varepsilon$ and by observing that $\Gamma \sim \Gamma'$. □

Next are two results that allow us to reason on the typability of a sub-expression that occurs in the hole of an evaluation context and on the fact that such sub-expression can be replaced by another one having the same type.

Lemma B.8 (typability of subterms). *If $\mathcal{D}$ is a derivation of $\Delta; \Gamma \vdash \mathcal{E}[e] : t$, then there exist Γ_1, Γ_2, and s such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\mathcal{D}$ has a sub-derivation $\mathcal{D}'$ concluding $\Delta; \Gamma_1 \vdash e : s$ and the position of $\mathcal{D}'$ in $\mathcal{D}$ corresponds to the position of $[\,]$ in $\mathcal{E}$.*

Proof. By induction on $\mathcal{E}$. We only discuss two cases, the others being analogous.

Case $\mathcal{E} = [\,]$. We conclude by taking $\Gamma_1 \stackrel{\text{def}}{=} \Gamma$, $\Gamma_2 \stackrel{\text{def}}{=} \emptyset$, and $s \stackrel{\text{def}}{=} t$. The sought sub-derivation $\mathcal{D}'$ is $\mathcal{D}$ itself.

Case $\mathcal{E} = \mathcal{E}'\ e'$. From [T-APP] we deduce that there exist Γ_1', Γ_2', and t' such that $\Gamma = \Gamma_1' + \Gamma_2'$ and $\Delta; \Gamma_1' \vdash \mathcal{E}'[e] : t' \to^\kappa t$ and $\Delta; \Gamma_2' \vdash e' : t'$. Let $\mathcal{D}''$ be the sub-derivation concluding $\Delta; \Gamma_1' \vdash \mathcal{E}'[e] : t' \to^\kappa t$. By induction hypothesis there exist Γ_1, Γ_2'', and s such that $\Gamma_1' = \Gamma_1 + \Gamma_2''$ and $\mathcal{D}''$ has a sub-derivation $\mathcal{D}'$ concluding $\Delta; \Gamma_1 \vdash e : s$ such that the position of $\mathcal{D}'$ in $\mathcal{D}''$ corresponds to the

position of [ ] in $\mathcal{E}'$. We conclude by taking $\Gamma_2 \stackrel{\text{def}}{=} \Gamma_2' + \Gamma_2''$ and observing that $\mathcal{D}'$ is a sub-derivation of $\mathcal{D}$ and that its position within $\mathcal{D}$ corresponds to the position of [ ] in $\mathcal{E}$. □

Lemma B.9 (replacement). *If:*

(1) $\mathcal{D}$ *is a derivation of* $\Delta; \Gamma_1 + \Gamma_2 \vdash \mathcal{E}[e] : t$,
(2) $\mathcal{D}'$ *is a sub-derivation of* $\mathcal{D}$ *concluding that* $\Delta; \Gamma_2 \vdash e : s$,
(3) *the position of* $\mathcal{D}'$ *in* $\mathcal{D}$ *corresponds to the position of* [ ] *in* $\mathcal{E}$,
(4) $\Delta; \Gamma_3 \vdash e' : s$,
(5) $\Gamma_1 + \Gamma_3$ *is defined,*

then $\Delta; \Gamma_1 + \Gamma_3 \vdash \mathcal{E}[e'] : t$.

Proof. By induction on $\mathcal{E}$. □

B.3 Progress for expressions

We provide a syntactic characterization of those expressions that do not reduce further. These are not necessarily values, for expressions may contain redexes involving primitives that reduce only at the level of processes.

Theorem 4.6 (progress for expressions). *If* Γ *is ground and well formed and* $\Delta; \Gamma \vdash e : t$ *and* $e \not\longrightarrow$, *then either* e *is a value or* $e = \mathcal{E}[K\ v]$ *for some* $\mathcal{E}$, v, w, *and* K *generated by the grammar*

$$K ::= \texttt{fork}\ w \mid \texttt{create} \mid \texttt{\_send}\ w \mid \texttt{\_receive} \mid \texttt{select}\ w \mid \texttt{branch} \mid \texttt{close}$$

Proof. If e is a value there is nothing left to prove, so we assume that e is not a value and proceed by induction on the structure of e and by cases on its shape. We only discuss a few representative cases, the others being simpler or similar.

Case $e = x$. This case is impossible because Γ is ground.

Case $e = e_1\ e_2$. Then $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_1 \vdash e_1 : s \to^{\kappa} t$ and $\Delta; \Gamma_2 \vdash e_2 : s$ where $\Gamma_1 :: \kappa$. We further distinguish three subcases. If e_1 is not a value, then by induction hypothesis we deduce that $e_1 = \mathcal{E}'[K\ v]$ for some $\mathcal{E}'$, K, and v and we conclude by taking $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{E}'\ e_2$. If e_1 is a value but e_2 is not, then by induction hypothesis we deduce that $e_2 = \mathcal{E}'[K\ v]$ for some $\mathcal{E}'$, K, and v and we conclude by taking $\mathcal{E} \stackrel{\text{def}}{=} e_1\ \mathcal{E}'$. If both e_1 and e_2 are values, we deduce that:

- e_1 cannot be an abstraction or $\texttt{fix}\ v$, for otherwise e would reduce;
- e_1 cannot be an identity abstraction or endpoint or package or $()$, $\texttt{C}\ v$, (v, w) because it has an arrow type;
- e_1 cannot be $\texttt{fix}$, $\texttt{fork}$, $\texttt{pair}$, $\texttt{C}$, $\texttt{\_send}$, $\texttt{select}$, $\texttt{pair}\ v$ or else e would be a value.

Then, e_1 must be one of $\texttt{create}$, $\texttt{\_receive}$, $\texttt{branch}$, $\texttt{fork}\ w$, $\texttt{\_send}\ w$, $\texttt{select}\ w$, or $\texttt{close}$. We conclude by taking $\mathcal{E} \stackrel{\text{def}}{=} [\ ]$, $K \stackrel{\text{def}}{=} e_1$, and $v \stackrel{\text{def}}{=} e_2$.

Case $e = \texttt{let}\ x, y = e_1\ \texttt{in}\ e_2$. Then $\Delta; \Gamma' \vdash e_1 : t \times s$ for some Γ' ground and $e_1 \not\longrightarrow$. Also, e_1 cannot be a value for the only values of type $t \times s$ are pairs whereas we know that e does not reduce. By induction hypothesis we deduce that $e_1 = \mathcal{E}'[K\ v]$ for some $\mathcal{E}'$, K, and v. We conclude by taking $\mathcal{E} \stackrel{\text{def}}{=} \texttt{let}\ x, y = \mathcal{E}'\ \texttt{in}\ e_2$.

Case $e = \{e'\}_{\varepsilon}$. Then $\Gamma = \Gamma', \varepsilon : [T; S]_{\varepsilon}$ and $\Delta; \Gamma', \varepsilon : [T]_{\varepsilon} \vdash e' : t \times [\mathbf{1}]_{\varepsilon}$ for some $e' \not\longrightarrow$ and t. If e' is not a value, then by induction hypothesis we deduce that $e' = \mathcal{E}'[K\ v]$ for some $\mathcal{E}'$, K, and v and we conclude by taking $\mathcal{E} \stackrel{\text{def}}{=} \{\mathcal{E}'\}_{\varepsilon}$. If e' is a value, then $e' = (v, w)$ for some v and w, for those are the only values with a product type. Further, it must be the case that $w = \varepsilon$, for endpoints are the only values that can have type $[\mathbf{1}]_{\varepsilon}$ and Γ is well formed by hypothesis. This is absurd because e does not reduce, hence this case is impossible. □

B.4 Subject reduction for processes

The following results rely on an easy weakening property concerning the identity environment:

Lemma B.10. *The following properties hold:*
(1) *If* $\Delta;\Gamma \vdash e : t$, *then* $\Delta,\Delta';\Gamma \vdash e : t$;
(2) *If* $\Delta;\Gamma \vdash P$, *then* $\Delta,\Delta';\Gamma \vdash P$.

Proof. By an easy induction on the respective derivations. □

The next result states the property that typing of processes is preserved by structural congruence.

Lemma B.11. *If* $\Delta;\Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Delta;\Gamma \vdash Q$.

Proof. An easy induction on the derivation of $P \equiv Q$ and by cases on the last rule applied. □

Theorem 4.7. *Let* $\Delta;\Gamma \vdash P$ *where* Γ *is ground, well formed and balanced. If* $P \xrightarrow{\ell} Q$, *then* $\Delta;\Gamma' \vdash Q$ *for some* Γ' *such that* $\Gamma \overset{\ell}{\Longrightarrow} \Gamma'$.

Proof. By induction on the derivation of $P \xrightarrow{\ell} Q$ and by cases on the last rule applied. We only discuss a few representative cases.

Case [R8]. Then $P = \langle \mathcal{E}[\texttt{fork}\ v\ w] \rangle \xrightarrow{\tau} \langle \mathcal{E}[()] \rangle \mid \langle v\ w \rangle = Q$. From [T-THREAD] we deduce $\Delta;\Gamma \vdash \mathcal{E}[\texttt{fork}\ v\ w] : \texttt{unit}$. From Lemma B.8 we deduce that there exist Γ_1, Γ_2, and t such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta;\Gamma_1 \vdash \texttt{fork}\ v\ w : t$. From [T-APP] and the type scheme of fork we deduce that there exist s, Γ_{11}, Γ_{12}, and Γ_{13} such that $\Delta;\Gamma_{11} \vdash \texttt{fork} : (s \to \texttt{unit}) \to s \to \texttt{unit}$ and $\Delta;\Gamma_{12} \vdash v : s \to \texttt{unit}$ and $t = \texttt{unit}$ and $\Delta;\Gamma_{13} \vdash w : s$. From [T-CONST] we deduce $\Gamma_{11} :: \mathsf{U}$. From Lemma B.9 and Lemma B.5 we derive $\Delta;\Gamma_{11} + \Gamma_2 \vdash \mathcal{E}[()] : \texttt{unit}$. Using [T-APP] we derive $\Delta;\Gamma_{12} + \Gamma_{13} \vdash v\ w : \texttt{unit}$. We conclude with two applications of [T-THREAD], one application of [T-PAR], and taking $\Gamma' \overset{\text{def}}{=} \Gamma$.

Case [R9]. Then $P = \langle \mathcal{E}[\texttt{create}\ ()] \rangle \xrightarrow{\tau} (\nu a)\langle \mathcal{E}[\lceil a, (a, \overline{a}) \rceil] \rangle = Q$ where a is fresh. From [T-THREAD] we deduce $\Delta;\Gamma \vdash \mathcal{E}[\texttt{create}\ ()] : \texttt{unit}$. From Lemma B.8 we deduce that there exist Γ_1, Γ_2, and t such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta;\Gamma_1 \vdash \texttt{create}\ () : t$. From [T-APP] and the type scheme of create we deduce that there exist Γ_{11} and Γ_{12} such that $\Delta;\Gamma_{11} \vdash \texttt{create} : \texttt{unit} \to t$ and $\Delta;\Gamma_{12} \vdash () : \texttt{unit}$ where $t = \exists \varrho.([T]_\varrho \times [\overline{T}]_{\overline{\varrho}})$ for some T and $\mathrm{fid}(t) \subseteq \Delta$. In particular, $a, \overline{a} \notin \mathrm{fid}(t)$. Using [T-NAME] and [T-APP] we derive $\Delta, a, \overline{a}; a : [T]_a, \overline{a} : [\overline{T}]_{\overline{a}} \vdash (a, \overline{a}) : [T]_a \times [\overline{T}]_{\overline{a}}$. Using [T-PACK] we derive $\Delta, a, \overline{a}; a : [T]_a, \overline{a} : [\overline{T}]_{\overline{a}} \vdash \lceil a, (a, \overline{a}) \rceil : \exists \varrho.([T]_\varrho \times [\overline{T}]_{\overline{\varrho}})$. From Lemma B.9 and Lemma B.10 we derive $\Delta, a, \overline{a}; \Gamma, a : [T]_a, \overline{a} : [\overline{T}]_{\overline{a}} \vdash \mathcal{E}[\lceil a, (a, \overline{a}) \rceil] : \texttt{unit}$. We conclude with one application of [T-THREAD] and one application of [T-SESSION], recalling that protocol duality is an involution, that $\sim$ is reflexive, and taking $\Gamma' \overset{\text{def}}{=} \Gamma$.

Case [R10]. Then $P = \langle \mathcal{E}[\texttt{\_send}\ v\ \varepsilon] \rangle \mid \langle \mathcal{E}'[\texttt{\_receive}\ \overline{\varepsilon}] \rangle \xrightarrow{\varepsilon} \langle \mathcal{E}[\varepsilon] \rangle \mid \langle \mathcal{E}'[(v, \overline{\varepsilon})] \rangle = Q$. From [T-PAR] and [T-THREAD] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta;\Gamma_1 \vdash \mathcal{E}[\texttt{\_send}\ v\ \varepsilon] : \texttt{unit}$ and $\Delta;\Gamma_2 \vdash \mathcal{E}'[\texttt{\_receive}\ \overline{\varepsilon}] : \texttt{unit}$. From the hypothesis that Γ is well formed, from Lemma B.8, [T-APP] and the type scheme of _send we deduce the following facts:

- $\Gamma_1 = \Gamma_{10} + \Gamma_{11} + \Gamma_{12} + \Gamma_{13}$;
- $\Delta;\Gamma_{11} \vdash \texttt{\_send} : t \to [!t]_\varepsilon \to^\kappa [\mathbf{1}]_\varepsilon$ where $\Gamma_{11} :: \mathsf{U}$ and $t :: \kappa$;
- $\Delta;\Gamma_{12} \vdash v : t$;
- $\Delta;\Gamma_{13} \vdash \varepsilon : [!t]_\varepsilon$, hence $\Gamma_{13} = \Gamma''_{13}, \varepsilon : [!t]_\varepsilon$ where $\Gamma''_{13} :: \mathsf{U}$.

From the hypothesis that Γ is well formed, from Lemma B.8, [T-APP] and the type scheme of _receive we deduce the following facts:

- $\Gamma_2 = \Gamma_{20} + \Gamma_{21} + \Gamma_{22}$;
- $\Delta; \Gamma_{21} \vdash \texttt{\_receive} : [?s]_{\overline{\varepsilon}} \to s \times [\mathbf{1}]_{\overline{\varepsilon}}$ where $\Gamma_{21} :: \mathsf{U}$;
- $\Delta; \Gamma_{22} \vdash \overline{\varepsilon} : [?s]_{\overline{\varepsilon}}$, hence $\Gamma_{22} = \Gamma''_{22}, \overline{\varepsilon} : [?s]_{\overline{\varepsilon}}$ where $\Gamma''_{22} :: \mathsf{U}$.

From the hypothesis that Γ is balanced we deduce that $!t \sim \overline{?s} = !s;$, that is $t = s$. Let $\Gamma'_{13} \stackrel{\text{def}}{=} \Gamma''_{13}, \varepsilon : [\mathbf{1}]_{\varepsilon}$ and $\Gamma'_{22} \stackrel{\text{def}}{=} \Gamma''_{22}, \overline{\varepsilon} : [\mathbf{1}]_{\overline{\varepsilon}}$ and let $\Gamma'_1 \stackrel{\text{def}}{=} \Gamma_{10} + \Gamma_{11} + \Gamma'_{13}$ and $\Gamma'_2 \stackrel{\text{def}}{=} \Gamma_{20} + \Gamma_{21} + \Gamma_{12} + \Gamma'_{22}$. From Lemma B.9 and Lemma B.5 we deduce $\Delta; \Gamma'_1 \vdash \mathcal{E}[\varepsilon] : \texttt{unit}$ and $\Delta; \Gamma'_2 \vdash \mathcal{E}'[(v, \overline{\varepsilon})] : \texttt{unit}$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma'_1 + \Gamma'_2$. We derive $\Delta; \Gamma' \vdash Q$ with one application of [T-PAR] and we conclude by observing that $\Gamma \xmapsto{\varepsilon} \Gamma'$.

Case [R11]. Then $P = \langle \mathcal{E}[\texttt{select}\ v\ \varepsilon] \rangle \mid \langle \mathcal{E}'[\texttt{branch}\ \overline{\varepsilon}] \rangle \xrightarrow{\varepsilon} \langle \mathcal{E}[\varepsilon] \rangle \mid \langle \mathcal{E}'[v\ \overline{\varepsilon}] \rangle = Q$. From [T-PAR] and [T-THREAD] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_1 \vdash \mathcal{E}[\texttt{select}\ v\ \varepsilon] : \texttt{unit}$ and $\Delta; \Gamma_2 \vdash \mathcal{E}'[\texttt{branch}\ \overline{\varepsilon}] : \texttt{unit}$. From the hypothesis that Γ is well formed, from Lemma B.8, [T-APP] and the type scheme of `select` we deduce the following facts:

- $\Gamma_1 = \Gamma_{10} + \Gamma_{11} + \Gamma_{12} + \Gamma_{13}$;
- $\Delta; \Gamma_{11} \vdash \texttt{select} : ([\overline{T_j}]_{\overline{\varepsilon}} \to^{\kappa} \{\mathtt{C}_i\ \texttt{of}\ [\overline{T_i}]_{\overline{\varepsilon}}\}_{i \in I}) \to [\oplus[\mathtt{C}_i : T_i]_{i \in I}]_{\varepsilon} \to^{\kappa} [T_j]_{\varepsilon}$ where $j \in I$ and $\Gamma_{11} :: \mathsf{U}$ and $t :: \kappa$;
- $\Delta; \Gamma_{12} \vdash v : [\overline{T_j}]_{\overline{\varepsilon}} \to^{\kappa} \{\mathtt{C}_i\ \texttt{of}\ [\overline{T_i}]_{\overline{\varepsilon}}\}_{i \in I}$;
- $\Delta; \Gamma_{13} \vdash \varepsilon : [\oplus[\mathtt{C}_i : T_i]_{i \in I}]_{\varepsilon}$, hence $\Gamma_{13} = \Gamma''_{13}, \varepsilon : [\oplus[\mathtt{C}_i : T_i]_{i \in I}]_{\varepsilon}$ where $\Gamma''_{13} :: \mathsf{U}$.

From the hypothesis that Γ is well formed, from Lemma B.8, [T-APP] and the type scheme of `branch` we deduce the following facts:

- $\Gamma_2 = \Gamma_{20} + \Gamma_{21} + \Gamma_{22}$;
- $\Delta; \Gamma_{21} \vdash \texttt{branch} : [\&[\mathtt{C}_i : S_i]_{i \in J}]_{\overline{\varepsilon}} \to \{\mathtt{C}_i\ \texttt{of}\ [S_i]_{\overline{\varepsilon}}\}_{i \in J}$ where $\Gamma_{21} :: \mathsf{U}$;
- $\Delta; \Gamma_{22} \vdash \overline{\varepsilon} : [\&[\mathtt{C}_i : S_i]_{i \in J}]_{\overline{\varepsilon}}$, hence $\Gamma_{22} = \Gamma''_{22}, \overline{\varepsilon} : [\&[\mathtt{C}_i : S_i]_{i \in J}]_{\overline{\varepsilon}}$ where $\Gamma''_{22} :: \mathsf{U}$.

From the hypothesis that Γ is balanced we deduce that $I = J$ and $T_i \sim \overline{S_i}$ for every $i \in I$. In particular, $T_j \sim \overline{S_j}$, that is $\overline{T_j} \sim S_j$. Let $\Gamma'_{13} \stackrel{\text{def}}{=} \Gamma''_{13}, \varepsilon : [T_j]_{\varepsilon}$ and $\Gamma'_{22} \stackrel{\text{def}}{=} \Gamma''_{22}, \overline{\varepsilon} : [\overline{T_j}]_{\overline{\varepsilon}}$ and let $\Gamma'_1 \stackrel{\text{def}}{=} \Gamma_{10} + \Gamma_{11} + \Gamma'_{13}$ and $\Gamma'_2 \stackrel{\text{def}}{=} \Gamma_{20} + \Gamma_{21} + \Gamma_{12} + \Gamma'_{22}$. From Lemma B.9, Lemma B.5, and [T-APP] we deduce $\Delta; \Gamma'_1 \vdash \mathcal{E}[\varepsilon] : \texttt{unit}$ and $\Delta; \Gamma'_2 \vdash \mathcal{E}'[v\ \overline{\varepsilon}] : \texttt{unit}$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma'_1 + \Gamma'_2$. We derive $\Delta; \Gamma' \vdash Q$ with one application of [T-PAR] and we conclude by observing that $\Gamma \xmapsto{\varepsilon} \xmapsto{\tau} \Gamma'$, where $\xmapsto{\varepsilon}$ accounts for the actual choice, whereas $\xmapsto{\tau}$ accounts for $\overline{T_j} \sim S_j$.

Case [R12]. Then $P = \langle \mathcal{E}[e] \rangle \xrightarrow{\tau} \langle \mathcal{E}[e'] \rangle = Q$ where $e \longrightarrow e'$. From [T-THREAD] we deduce $\Delta; \Gamma \vdash \mathcal{E}[e] : \texttt{unit}$. From Lemma B.8 we deduce that there exist Γ_1, Γ_2, and t such that $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_1 \vdash e : t$. From Theorem 4.5 we deduce $\Delta; \Gamma'_1 \vdash e' : t$ for some $\Gamma'_1 \sim \Gamma_1$. From Lemma B.9 and using one application of [R-THREAD] we derive $\Delta; \Gamma'_1 + \Gamma_2 \vdash \mathcal{E}[e']$. We conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma'_1 + \Gamma_2$ and observing that $\Gamma \xmapsto{\tau} \Gamma'$.

Case [R13]. Then $P = P_1 \mid P_2 \xrightarrow{\ell} P'_1 \mid P_2 = Q$ where $P_1 \xrightarrow{\ell} P'_1$. From [T-PAR] we deduce $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta; \Gamma_i \vdash P_i$ for $i = 1, 2$. By induction hypothesis we derive $\Delta; \Gamma'_1 \vdash P'_1$ for some Γ'_1 such that $\Gamma_1 \stackrel{\ell}{\Longmapsto} \Gamma'_1$. We conclude by taking $\Gamma' \stackrel{\text{def}}{=} \Gamma'_1 + \Gamma_2$ using one application of [T-PAR] and Proposition B.1.

Case [R14]. Then $P = (\nu a)P' \xrightarrow{\ell} (\nu a)Q' = Q$ where $P' \xrightarrow{\ell} Q'$ and $\ell \notin \{a, \overline{a}\}$. From [T-SESSION] we deduce $\Delta, a, \overline{a}; \Gamma, a : [T]_a, \overline{a} : [S]_{\overline{a}} \vdash P'$ and $T \sim \overline{S}$. Note that $\Gamma, a : [T]_a, \overline{a} : [S]_{\overline{a}}$ is well formed, ground and balanced if so is Γ. By induction hypothesis we deduce that there exist Γ', T', and S' such that $\Delta, a, \overline{a}; \Gamma', a : [T']_a, \overline{a} : [S']_{\overline{a}} \vdash Q'$ and $\Gamma, a : [T]_a, \overline{a} : [S]_{\overline{a}} \stackrel{\ell}{\Longmapsto} \Gamma', a : [T']_a, \overline{a} : [S']_{\overline{a}}$, hence

$\Gamma \overset{\ell}{\Longmapsto} \Gamma'$ using the fact that $\ell \notin \{a, \overline{a}\}$. From Proposition B.1 we deduce that $T' \sim \overline{S'}$. We conclude with one application of [T-SESSION].

Case [R15]. Similar to the previous case.

Case [R16]. Easy consequence of Lemma B.11 and the induction hypothesis. □

C OCAML ENCODING OF SESSION TYPES

We only prove Theorem 5.2. The proof of Theorem 5.5 is essentially the same.

THEOREM 5.2. *If* $[\![T]\!] = (\tau_1, \tau_2)$ `t`, *then* $[\![\overline{T}]\!] = (\tau_2, \tau_1)$ `t`.

PROOF. By cases on the structure of T. We omit trivial and symmetric cases.

Case $T = ?t$. Then $[\![T]\!] = (\langle\!\langle t \rangle\!\rangle$ `msg,0) t` and we conclude by observing that $[\![\overline{T}]\!] = [\![!t]\!] =$ `(0,`$\langle\!\langle t \rangle\!\rangle$ `msg) t`.

Case $T = \&[\mathtt{C}_i : T_i]_{i \in I}$. Then $[\![\&[\mathtt{C}_i : T_i]_{i \in I}]\!] =$ `({`$\mathtt{C}_i$ `of` $[\![T_i]\!]\}_{i \in I}$ `tag,0) t` and we conclude by observing that $[\![\overline{T}]\!] = [\![\oplus[\mathtt{C}_i : \overline{T}_i]_{i \in I}]\!] =$ `(0,{`$\mathtt{C}_i$ `of` $[\![\overline{T_i}]\!]\}_{i \in I}$ `tag) t = (0,{`$\mathtt{C}_i$ `of` $[\![T_i]\!]\}_{i \in I}$ `tag) t` using the definition of duality, Definition 5.1, and the fact that duality is an involution.

Case $T = T_1; T_2$. Then $[\![T]\!] =$ `((`$[\![T_1]\!], [\![T_2]\!]$`) seq, (`$[\![\overline{T_1}]\!], [\![\overline{T_2}]\!]$`) seq) t` and we conclude by observing that $[\![\overline{T}]\!] =$ `((`$[\![\overline{T_1}]\!], [\![\overline{T_2}]\!]$`) seq, (`$[\![\overline{\overline{T_1}}]\!], [\![\overline{\overline{T_2}}]\!]$`) seq) t = ((`$[\![\overline{T_1}]\!], [\![\overline{T_2}]\!]$`) seq, (`$[\![T_1]\!], [\![T_2]\!]$`) seq) t` using the definition of duality, Definition 5.1, and the fact that duality is an involution. □

Received April 2017; revised January 2018; accepted May 2018