Chaperone Contracts for Higher-Order Sessions

Chaperone Contracts for Higher-Order Sessions

HERNÁN MELGRATTI, Universidad de Buenos Aires, FCEyN and CONICET-UBA, ICC
LUCA PADOVANI, Università di Torino

Contracts have proved to be an effective mechanism that helps developers in identifying those modules of a program that violate the contracts of the functions and objects they use. In recent years, sessions have been established as a key mechanism for realizing inter-module communications in concurrent programs. Just like values flow into or out of a function or object, messages are sent on, and received from, a session endpoint. Unlike conventional functions and objects, however, the kind, direction, and properties of messages exchanged in a session may vary over time, as the session progresses. This feature of sessions calls for contracts that evolve along with the session they describe.

In this work, we extend to sessions the notion of chaperone contract (roughly, a contract that applies to a mutable object) and investigate the ramifications of contract monitoring in a higher-order language that features sessions. We give a characterization of a correct module, one that honors the contracts of the sessions it uses, and prove a blame theorem. Guided by the calculus, we describe a lightweight implementation of monitored sessions as an `OCaml` module with which programmers can benefit from static session type checking and dynamic contract monitoring using an off-the-shelf version of `OCaml`.

CCS Concepts: • **Theory of computation → Program specifications**; **Pre- and post-conditions**; *Type structures*; • **Software and its engineering → Constraints**; **Concurrent programming structures**;

Additional Key Words and Phrases: Chaperone contracts, sessions, blame soundness, `OCaml`

ACM Reference format:
Hernán Melgratti and Luca Padovani. 2017. Chaperone Contracts for Higher-Order Sessions. *Proc. ACM Program. Lang.* 1, 1, Article 35 (September 2017), 29 pages.
https://doi.org/10.1145/3110279

1 INTRODUCTION

The design-by-contract approach to software development [Meyer 1992] promotes the usage of executable specifications called *contracts* to describe the mutual obligations that regulate the interaction between different modules. Contracts are embedded in code and checked at runtime to help developers identify faulty modules. Findler and Felleisen [2002] have shown that this approach is applicable also to languages with higher-order functions, despite the fact that checking whether a function satisfies a given contract is undecidable. The key idea is to defer the evaluation of the contract until the function is actually applied to its argument. At this point, the contract is disassembled and used to perform or delay checks on the supplied argument and the returned result. Specifying contracts for mutable objects poses additional problems, though, because checking whether an object satisfies a given contract at a particular point in time does not guarantee that the contract will be also satisfied later on, if the object is modified. To address these situations, Strickland et al. [2012] introduced *chaperone contracts*, an interposition mechanism whereby the object is protected by a proxy that exposes the same interface as the object and that takes actions to perform or delay checks on the values flowing into or out of the object.

2017. 2475-1421/2017/9-ART35 $15.00
https://doi.org/10.1145/3110279

In this paper we introduce chaperone contracts for *binary sessions*, which are private communication channels between pairs of processes. Just like mutable objects have operations for inserting and retrieving values, sessions have operations for sending and receiving messages. Unlike conventional objects, though, the *order* in which these operations can be performed is disciplined by a protocol specification called *session type*. Also, the type of exchanged messages may change over time. For these reasons, sessions are more closely related to *non-uniform objects* [Gay et al. 2010; Ravara and Vasconcelos 2000], whose interface changes according to their state. Chaperone contracts for sessions can be used to complement session types with precise specifications concerning the content of messages and their relationship with the actual behavior of session participants. Compared to the functional contracts of Findler and Felleisen [2002] and chaperone contracts of Strickland et al. [2012], the distinguishing feature of chaperone contracts for sessions is that they evolve as the monitored session progresses: not only the contracts of messages exchanged in the session may vary over time, but also the contract of the remainder of a session may depend upon messages that have been exchanged earlier on in the same or a different session.

More in detail, here are the main contributions of this work:

- We define a core functional language called λCoS featuring runtime contract monitoring for higher-order sessions in the style of Findler and Felleisen [2002] and Strickland et al. [2012]. Contracts are dynamically updated along with the session they monitor. We introduce a novel semantics for contract monitoring, called *big-step monitoring*, that is instrumental to the subsequent formal analysis of our monitoring system. The setting allows us to pinpoint unexplored aspects of contract monitoring in a language with linear resources.
- We give an operational characterization of *locally correct modules*, namely modules that honor the contracts of the sessions they use. We argue that this notion can help programmers write correct code. We also prove a *blame soundness* result stating that correct modules cannot be blamed even in the presence of buggy or malicious parties.
- Guided by λCoS, we implement chaperone contracts for sessions on top of an existing `OCaml` library for binary sessions [Padovani 2017]. The implementation also supports runtime contract monitoring in the presence of higher-order sessions and its key aspects are portable to other programming languages as well.

Although contracts for sessions and corresponding monitoring techniques have been actively investigated (*cf.* Section 7), our proposal is the first one that follows the approach of Findler and Felleisen [2002], which has well-known assets: contracts are first-class entities written in the same language programmers are already accustomed to, they can be computed, passed as arguments, and returned as results, they can be used gradually within large systems and can describe whole protocols or just parts thereof. Also, monitoring is performed inline and requires no external tools or dedicated middleware.

Structure of the paper. The next section gives an overview of the key ingredients of session contracts and their usage. Although the section is informal, the given examples are "real" in the sense that they run using our implementation. The formal model of λCoS is given in Section 3 and its typing discipline in Section 4. To keep the formalization technically manageable, we make some simplifying assumptions on the operational semantics and the type system which are relaxed in the implementation. Section 5 addresses local correctness and blame soundness. The implementation is detailed in Section 6, whereas Section 7 discusses connections with related work in more detail. Section 8 summarizes our work and hints at some extensions and future developments. Supplementary technical material and proofs of the results presented in Sections 3–6 can be found in the companion technical report [Melgratti and Padovani 2017].

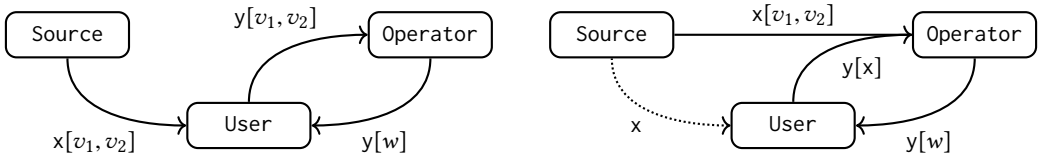

Fig. 1. Stream processing networks with first-order (left) and second-order (right) sessions.

2 A PROGRAMMER'S VIEWPOINT OF CONTRACTS FOR SESSIONS

In this section we provide a programmer's viewpoint of contracts for sessions using the process networks depicted in Figure 1 as running examples. In these networks, the aim of module `User` is to obtain an element w resulting from two elements v_1, v_2 produced by `Source` and combined by `Operator`. For the sake of illustration, we assume that all the elements are integer numbers and that $w = v_1$ `mod` v_2.

According to the network on the left-hand side of Figure 1, `User` establishes two sessions `x` and `y` with `Source` and `Operator` respectively, it forwards every element v_i received from `x` on `y`, and receives the transformed element w from `y`. Session types allow us to formalize these protocols specifying the type, direction and order of messages exchanged within sessions. In this case, the session endpoints `x` and `y` owned by `User` are typed as `x : ?int.?int.end` and `y : !int.!int.?int.end`. In words, `User` uses `x` to receive two integer numbers and `y` to send two numbers and then to receive another one. What session types do not describe are additional requirements and guarantees concerning the content of exchanged messages. In this scenario, for example, it could be sensible to specify that the second number sent from `User` to `Operator` should be different from zero and that the number sent from `Operator` to `User` is non negative. Such specifications – hereafter called *contracts* – could be used to monitor, at runtime, the interaction between `User` and `Operator` so as to detect contract violations and, hopefully, to guide programmers to the source of the problem.

The approach of Findler and Felleisen [2002] to contract monitoring rests on three key ingredients: (1) a set of combinators for writing contracts; (2) a mechanism for associating contracts with the entities being monitored; (3) a labeling of the modules involved that narrows the source of the problem in case a contract violation is detected. In our setting, the definition

```
let operator_chan = register operator_body operator_c "Operator"
```

registers a new service channel `operator_chan` that can be used to initiate sessions with `Operator`. Registration links together three pieces of information: the body of the process that handles each session initiated with `Operator` (`operator_body`, omitted here), the contract that `Operator` claims to satisfy (`operator_c`), and a symbolic label that identifies `Operator` (the string `"Operator"`). The contract `operator_c` is defined as

```
let operator_c =
 send_c any_c @@ send_c (flat_c (≠ 0)) @@ receive_c (flat_c (≥ 0)) @@ end_c
```

and its structure reflects that of the protocol `!int.!int.?int.end` of the session it describes. Let us focus first on the four sub-contracts separated by `@@` and built with the `send_c`, `receive_c`, and `end_c` combinators. As their name suggest, each of these indicates either an output or an input or the termination of the session. Both `send_c` and `receive_c` have an argument which is itself a contract describing the exchanged message: depending on whether the operation is an input or an output, the contract specifies a requirement or a guarantee for the message. So, the contract

`flat_c (≠ 0)` is satisfied by any non-zero integer, `flat_c (≥ 0)` is satisfied by any non-negative integer, and `any_c` is satisfied by any value. Note that the contract is written from the viewpoint of a client of `Operator`, such as `User`. The combinator `@@` plays the same role as the dot '.' in session types and composes contracts sequentially: it indicates that the contract associated with the endpoint, and the specified constraints on the exchanged messages, vary after each interaction: `Operator` first accepts an arbitrary integer number, it then expects a non-zero number from the client, and it finally sends back a non-negative number. As it turns out, `@@` is nothing but function application. Therefore, `operator_c` can be alternatively defined as

```
send_c any_c (send_c (flat_c (≠ 0)) (receive_c (flat_c (≥ 0)) end_c))
```

We will keep using `@@` in this section for the sake of readability. In the rest of the paper we will drop `@@` and use parentheses to disambiguate the structure of contracts when necessary.

The definitions

```
let source_c = receive_c any_c @@ receive_c any_c @@ end_c
let source_chan = register source_body source_c "Source"
```

register a service channel for `Source` in an analogous way. In this case, `source_c` does not specify any additional constraints with respect to the session type `?int.?int.end`. With `Source` and `Operator` in place, we are ready to implement `User`:

```
1  let user () =
2    let x = connect source_chan "User" in   (* connect with Source    *)
3    let y = connect operator_chan "User" in (* connect with Operator  *)
4    let v1, x = receive x in                (* receive v1 from Source *)
5    let v2, x = receive x in                (* receive v2 from Source *)
6    let y = send v1 y in                    (* send v1 to Operator    *)
7    let y = send v2 y in                    (* send v2 to Operator    *)
8    let w, y = receive y in                 (* receive result         *)
9    print_int w; close x; close y           (* close sessions         *)
```

After initiating the two sessions `x` and `y` (lines 2–3), `User` implements the behavior informally described above making use of the operations `send` and `receive` for session communications. The series of rebindings of the session endpoints `x` and `y` is a common trait of most implementations of sessions for functional languages and allows for the type of `x` and `y` to be appropriately updated after each operation. The `connect` primitive specifies the shared channel on which the sessions are initiated and a label that identifies the requesting party. This label, together with those specified at the time of service registration, is used by the runtime monitoring mechanism to pinpoint the source of a contract violation. Here, `"User"` identifies the module responsible for the messages sent on `x` and `y`, whereas `"Source"` and `"Operator"`, associated with `source_chan` and `operator_chan` at the time of their registration, identify the modules that are responsible for the messages received from `x` and `y`, respectively. So if v_2 turns out to be 0, the runtime monitoring of the `y` session endpoint flags `User` for breaching the contract with `Operator`. If, on the other hand, `Operator` sends a negative remainder back to `User`, then it is `Operator` that violates its own contract.

In this first example, the module responsible for a contract violation is easily identified with the sender of the message that triggers the violation. This is not always the case, though. Consider for instance the network on the right-hand side of Figure 1, which aims to achieve the same computation as before, except that `User` now *delegates* the session endpoint `x` to `Operator` so that `Operator` receives the elements v_1, v_2 directly from `Source`, rather than having them forwarded through `User`. In this case, the session type associated with `y` is `!(?int.?int.end).?int.end`, indicating that

User first sends Operator a session endpoint of type ?int.?int.end (that is precisely the type of x) and then behaves according to ?int.end to receive the result from Operator. Correspondingly, the contract exposed by Operator could be the following

```
let operator_deleg_c =
  send_c (receive_c any_c @@ receive_c (flat_c (≠ 0)) @@ end_c) @@
  receive_c (flat_c (≥ 0)) @@ end_c
```

specifying that the first message received by Operator is a session endpoint from which the second receive element is supposed to be different from zero. That is, operator_deleg_c is a second-order contract for a second-order session.

The implementation of User becomes

```
1 let user_deleg () =
2   let x = connect source_chan "User" in
3   let y = connect operator_deleg_chan "User" in
4   let y = send x y in
5   let res, y = receive y in
6   print_int res; close y
```

which establishes the two sessions as before (lines 2–3), delegates x on y (line 4), and then receives the result from Operator (line 5) before closing the session (line 6). As before, the arrival of a zero v_2 element to Operator should trigger a contract violation, but establishing which process is to blame is not as easy as in the previous example. On the one hand, the offending element is sent by Source so, if we were to follow the simple rule for blame assignment used before, we would conclude that Source is to blame. On the other hand, Source has never violated the source_c contract it was registered with. The problem lies once again within User: by delegating x on y, User is claiming that the contract source_c associated with x *entails* the contract

```
receive_c any_c @@ receive_c (flat_c (≠ 0)) @@ end_c
```

but this is not true, because not every message satisfying any_c also satisfies flat_c (≠ 0). In summary, the actual offending operation is the delegation performed by User, but establishing this fact would require a decision procedure for contract entailment, which is instead an undecidable relation (contracts may contain arbitrarily complex predicates expressed using a Turing complete language). Therefore, the delegation is provisionally accepted as valid and triggers a suitable rearrangement of contracts and labels for the involved endpoints such that, in the event of a contract violation, User and not Source is blamed.

In the rest of the paper we extend and formalize the ideas sketched in this section. In particular, we will consider a superset of the contract combinators used in this section that allow us to specify *dependent, possibly branching contracts* whose structure depends on the content of previously exchanged messages. The formalization will culminate with a blame soundness result stating that a process cannot be blamed by the monitor if it always respects the contracts of the endpoints it uses.

3 SYNTAX AND SEMANTICS OF λCoS

3.1 Syntax

We use infinite sets of *variables* x, y, z and service/session channels a, b, c. We use *polarities* $\iota \in \{+, -\}$ to distinguish the two endpoints of a session channel and write $\overline{\iota}$ for the dual polarity of ι, where $\overline{+} = -$ and $\overline{-} = +$. An *endpoint* is a pair a^ι made of a session channel a and a polarity ι and we say that $a^{\overline{\iota}}$ is the *peer* of a^ι. A *name* u is either a variable or a channel or an endpoint.

Table 1. Syntax of λCoS (terms that occur only at runtime and not in user code are shaded).

Process	P, Q	::=	$\langle e \rangle_p \mid a \Leftarrow_p^{\mathtt{c}} v \mid P \parallel Q \mid (\nu a)P$
Expression	e	::=	$v \mid x \mid e_1 e_2 \mid \mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2 \mid \mathtt{case}\ e\ \mathtt{of}\ e_1 \mid e_2$
		|	$[\![e_1]\!]^{e_2,p,q} \mid v \blacktriangleleft^p e \mid \mathtt{blame}\ p$
Value	$v, w, \mathtt{c}, \mathtt{d}$	::=	$\lambda x.e \mid \varepsilon$
		|	$\mathtt{c}\ v_1 \cdots v_n \qquad (0 \le n \le \#\mathtt{c})$
Endpoint	ε	::=	$a^\iota \mid [\![\varepsilon]\!]^{\mathtt{c},p,q}$
Constant	$\mathtt{c}$	::=	$() \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{inl} \mid \mathtt{inr} \mid \mathtt{pair} \mid \mathtt{dual}$
		|	$\mathtt{connect} \mid \mathtt{close} \mid \mathtt{send} \mid \mathtt{receive} \mid \mathtt{branch} \mid \mathtt{left} \mid \mathtt{right}$
		|	$\mathtt{flat\_c} \mid \mathtt{end\_c} \mid \mathtt{branch\_c} \mid \mathtt{choice\_c}$
		|	$\mathtt{send\_c} \mid \mathtt{send\_d} \mid \mathtt{receive\_c} \mid \mathtt{receive\_d}$

The syntax of λCoS comprises *processes* and *expressions* (Table 1). Expressions model the sequential part of programs and processes model parallel threads that invoke services and communicate through sessions. A process is either a *thread* $\langle e \rangle_p$ made of a body e and a label p, a *service* $a \Leftarrow_p^{\mathtt{c}} v$ that waits for invocations on the service channel a and spawns new threads with body v, the *parallel composition* $P \parallel Q$ of two processes P and Q, or a *session* $(\nu a)P$ with scope P. Each service advertises a contract $\mathtt{c}$ that describes the intended interaction from the client's viewpoint. For simplicity, we assume that services cannot be created dynamically in λCoS. As we have seen in Section 2, the implementation is more liberal and allows new services to be dynamically created and registered. A *module* is a set of threads and services with the same label. We use *labels* p, q to identify modules and to assign blame.

Expressions include the standard constructs of the λ-calculus, pair splitting $\mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2$, and pattern matching $\mathtt{case}\ e\ \mathtt{of}\ e_1 \mid e_2$. Additionally, we have three constructs that support runtime monitoring [Findler and Felleisen 2002; Wadler and Findler 2009]. A *monitored expression* $[\![e_1]\!]^{e_2,p,q}$ wraps e_1 with a contract e_2 and the labels p and q identify the modules responsible for the values respectively flowing out of and into e_1. Such modules may be blamed if the flowing values do not satisfy the constraints specified by the contract e_2. In λCoS, the intuition behind these flows of values is best illustrated by thinking of e_1 as a session endpoint. Then, p is the module responsible for the messages *received* from the endpoint, whereas q is the module responsible for the messages *sent* on the endpoint. Sometimes we use σ and ϱ to denote pairs of labels such as p, q, and we write $\neg\sigma$ for the symmetric pair obtained by swapping the components in σ. We say that p is the *positive* label and q the *negative* label in the pair p, q. Monitors may accumulate on top of each other yielding stacks of the form $[\![\cdots[\![e]\!]^{\mathtt{c}_1,\sigma_1}\cdots]\!]^{\mathtt{c}_n,\sigma_n}$ that we abbreviate as $[\![e]\!]^{\overrightarrow{\mathtt{c},\sigma}}$. Values flowing out of e are checked against all the contracts $\mathtt{c}_1, \ldots, \mathtt{c}_n$ starting from $\mathtt{c}_1$, whereas values flowing into e are checked against all the contracts $\mathtt{c}_n, \ldots, \mathtt{c}_1$ starting from $\mathtt{c}_n$. It is sometimes necessary to reverse the order of monitors while swapping the elements of the pairs σ_i. In these cases we write $[\![e]\!]^{\overleftarrow{\mathtt{c},\neg\sigma}}$ for $[\![\cdots[\![e]\!]^{\mathtt{c}_n,\neg\sigma_n}\cdots]\!]^{\mathtt{c}_1,\neg\sigma_1}$. A *busy monitor* $v \blacktriangleleft^p e$ indicates an ongoing check being performed on value v. If e evaluates to $\mathtt{true}$ then the check is passed and the busy monitor reduces to v. If e evaluates to $\mathtt{false}$ then the busy monitor reduces to $\mathtt{blame}\ p$ signaling a contract violation of p. So, a busy monitor $v \blacktriangleleft^p e$ is akin to a conditional $\mathtt{if}\ e\ \mathtt{then}\ v\ \mathtt{else}\ \mathtt{blame}\ p$.

We use v, w, $\mathtt{c}$ and $\mathtt{d}$ to range over *values*, which comprise abstractions, (monitored) endpoints, and (applied) constants. We reserve $\mathtt{c}$ and $\mathtt{d}$ for values that represent contracts, which are described below. Constants comprise some standard data constructors (unit, booleans, pairs, sums) and a

Table 2. Reduction of expressions (contextual rule omitted).

[R1]	$(\lambda x.e)v \longrightarrow e\{v/x\}$	[R8]	$\texttt{dual end\_c} \longrightarrow \texttt{end\_c}$
[R2]	$\texttt{let } x,y = (v,w) \texttt{ in } e \longrightarrow e\{v,w/x,y\}$	[R9]	$\texttt{dual } !\mathrm{C};\mathrm{D} \longrightarrow ?\mathrm{C};\texttt{dual } \mathrm{D}$
[R3]	$\texttt{case inl } v \texttt{ of } e_1 \mid e_2 \longrightarrow e_1 v$	[R10]	$\texttt{dual } ?\mathrm{C};\mathrm{D} \longrightarrow !\mathrm{C};\texttt{dual } \mathrm{D}$
[R4]	$\texttt{case inr } v \texttt{ of } e_1 \mid e_2 \longrightarrow e_2 v$	[R11]	$\texttt{dual } !\mathrm{C}.w \longrightarrow ?\mathrm{C}.\lambda x.\texttt{dual } (wx)$
[R5]	$[v]^{\texttt{flat\_c } w,p,q} \longrightarrow v \triangleleft^p wv$	[R12]	$\texttt{dual } ?\mathrm{C}.w \longrightarrow !\mathrm{C}.\lambda x.\texttt{dual } (wx)$
[R6]	$v \triangleleft^p \texttt{true} \longrightarrow v$	[R13]	$\texttt{dual } !\mathrm{C}.\mathrm{D}:\mathrm{E} \longrightarrow ?\mathrm{C}.\texttt{dual } \mathrm{D}:\texttt{dual } \mathrm{E}$
[R7]	$v \triangleleft^p \texttt{false} \longrightarrow \texttt{blame } p$	[R14]	$\texttt{dual } ?\mathrm{C}.\mathrm{D}:\mathrm{E} \longrightarrow !\mathrm{C}.\texttt{dual } \mathrm{D}:\texttt{dual } \mathrm{E}$

standard set of session primitives for connecting to services (`connect`), sending and receiving messages (`send` and `receive`), selecting a choice (`left` and `right`) and offering a choice (`branch`).

Constants of the form $*$`_c` and $*$`_d` are *contract constructors*. A *flat* contract `flat_c` w is satisfied by every value v such that wv evaluates to `true`. We say that w is the *predicate* of the contract `flat_c` w.

The contract `end_c` describes an endpoint that can only be closed. The *non-dependent contracts* `send_c` C D and `receive_c` C D (sugared as $!\mathrm{C};\mathrm{D}$ and $?\mathrm{C};\mathrm{D}$) have a prefix C and a continuation D and describe endpoints used for respectively sending and receiving a message that satisfies C and then used according to D. These contracts are qualified as "non dependent" because the continuation contract D does *not* depend on the exchanged message. The *dependent contracts* `send_d` C w and `receive_d` C w (sugared as $!\mathrm{C}.w$ and $?\mathrm{C}.w$) describe endpoints used for respectively sending and receiving a message v that satisfies C and then used according to wv. That is, w is a function that produces the continuation contract when applied to the exchanged message. In principle, a non-dependent contract such as $!\mathrm{C};\mathrm{D}$ could be treated as a degenerate dependent contract $!\mathrm{C}.\lambda\_.\mathrm{D}$, where the function that computes the continuation contract is constant. However, the two contract forms require different typing policies and therefore must be kept distinct, at least in the formal model. In the implementation, which is based on a weaker type system, the two forms can be unified so that non-dependent contracts are indeed degenerate cases of dependent ones. The contracts `choice_c` C D E and `branch_c` C D E (sugared as $!\mathrm{C}.\mathrm{D}:\mathrm{E}$ and $?\mathrm{C}.\mathrm{D}:\mathrm{E}$) describe endpoints used for respectively selecting and offering a choice. The choice is effectively represented and transmitted as a boolean value v that satisfies C. The continuation contracts D and E describe the endpoint after the choice, depending on whether v is `true` or `false`. Compared to dependent contracts, choices also allow the session types of the continuations, and not just the contracts, to depend on the exchanged message. There is a key difference between flat and session contracts. Flat contracts are meant to be checked right away, turning a monitor into a busy monitor. Session contracts chaperone the endpoints they wrap and are checked only if and when the endpoints are used for input/output operations. Also, session contracts are dynamically updated as the endpoint they wrap is used.

The primitive `dual` computes the *dual* of a contract C, which specifies the same constraints as C except for the direction of messages, which is reversed. We will see how it is used later on.

An applied constant `c` $v_1 \cdots v_n$ is a value only if $0 \le n \le \#\texttt{c}$ where `#c` denotes the arity of `c`, that is the maximum number of arguments to which `c` can be applied before possibly becoming a redex. In λCoS, `branch_c` and `choice_c` have arity 3, `pair`, `send_c`, `receive_c`, `send_d` and `receive_d` have arity 2, `inl`, `inr`, `send` and `flat_c` have arity 1, and all the other constants have arity 0.

We identify processes and expressions modulo α-renaming of bound names, considering that $(\nu a)P$ binds a, a^+, and a^- in P, and we assume that bound names are all distinct. We say that P is a *user process* if it does not use any runtime syntax as by Table 1.

Table 3. Reduction of processes (contextual rules omitted).

$$\text{[R15]}\quad \begin{pmatrix}\langle \mathscr{E}[\texttt{connect}\ a]\rangle_p \\ a \Leftarrow^{\mathrm{C}}_q v\end{pmatrix} \to (\nu b)\begin{pmatrix}\langle \mathscr{E}[[b^+]^{\mathrm{C},q,p}]\rangle_p \\ \langle v\ [b^-]^{\texttt{dual}\ \mathrm{C},p,q}\rangle_q\end{pmatrix} \parallel a \Leftarrow^{\mathrm{C}}_q v \qquad b \text{ fresh}$$

$$\text{[R16]}\quad (\nu a)\begin{pmatrix}\langle \mathscr{E}[\texttt{close}\ [a^+]^{\overrightarrow{\texttt{end\_c},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[\texttt{close}\ [a^-]^{\overrightarrow{\texttt{end\_c},\varrho}}]\rangle_q\end{pmatrix} \to \langle \mathscr{E}[()]\rangle_p \parallel \langle \mathscr{E}'[()]\rangle_q$$

$$\text{[R17]}\quad \begin{pmatrix}\langle \mathscr{E}[\texttt{send}\ v\ [a^\iota]^{\overrightarrow{!\mathrm{C};\mathrm{D},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[\texttt{receive}\ [a^{\bar\iota}]^{\overrightarrow{?\mathrm{E};\mathrm{F},\varrho}}]\rangle_q\end{pmatrix} \to \begin{pmatrix}\langle \mathscr{E}[[a^\iota]^{\overrightarrow{\mathrm{D},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[([[v]^{\overleftarrow{\mathrm{C},\neg\sigma}}]^{\overrightarrow{\mathrm{E},\varrho}}, [a^{\bar\iota}]^{\overrightarrow{\mathrm{F},\varrho}})]\rangle_q\end{pmatrix}$$

$$\text{[R18]}\quad \begin{pmatrix}\langle \mathscr{E}[\texttt{send}\ v\ [a^\iota]^{\overrightarrow{!\mathrm{C}\cdot w_1,\sigma}}]\rangle_p \\ \langle \mathscr{E}'[\texttt{receive}\ [a^{\bar\iota}]^{\overrightarrow{?\mathrm{D}\cdot w_2,\varrho}}]\rangle_q\end{pmatrix} \to \begin{pmatrix}\langle \mathscr{E}[[a^\iota]^{\overrightarrow{w_1 v,\sigma}}]\rangle_p \\ \langle \mathscr{E}'[([[v]^{\overleftarrow{\mathrm{C},\neg\sigma}}]^{\overrightarrow{\mathrm{D},\varrho}}, [a^{\bar\iota}]^{\overrightarrow{w_2 v,\varrho}})]\rangle_q\end{pmatrix}$$

$$\text{[R19]}\quad \begin{pmatrix}\langle \mathscr{E}[\texttt{left}\ [a^\iota]^{\overrightarrow{!\mathrm{C}\cdot\mathrm{D}:\mathrm{E},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[\texttt{branch}\ [a^{\bar\iota}]^{\overrightarrow{?\mathrm{F}\cdot\mathrm{G}:\mathrm{H},\varrho}}]\rangle_q\end{pmatrix} \to \begin{pmatrix}\langle \mathscr{E}[[a^\iota]^{\overrightarrow{\mathrm{D},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[(\lambda\_.\,\texttt{inl}\ [a^{\bar\iota}]^{\overrightarrow{\mathrm{G},\varrho}})\ [[\texttt{true}]^{\overleftarrow{\mathrm{C},\neg\sigma}}]^{\overrightarrow{\mathrm{F},\varrho}}]\rangle_q\end{pmatrix}$$

$$\text{[R20]}\quad \begin{pmatrix}\langle \mathscr{E}[\texttt{right}\ [a^\iota]^{\overrightarrow{!\mathrm{C}\cdot\mathrm{D}:\mathrm{E},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[\texttt{branch}\ [a^{\bar\iota}]^{\overrightarrow{?\mathrm{F}\cdot\mathrm{G}:\mathrm{H},\varrho}}]\rangle_q\end{pmatrix} \to \begin{pmatrix}\langle \mathscr{E}[[a^\iota]^{\overrightarrow{\mathrm{E},\sigma}}]\rangle_p \\ \langle \mathscr{E}'[(\lambda\_.\,\texttt{inr}\ [a^{\bar\iota}]^{\overrightarrow{\mathrm{H},\varrho}})\ [[\texttt{false}]^{\overleftarrow{\mathrm{C},\neg\sigma}}]^{\overrightarrow{\mathrm{F},\varrho}}]\rangle_q\end{pmatrix}$$

3.2 Semantics

Expressions reduce according to a call-by-value strategy, for which we define *evaluation contexts* thus:

$$\mathscr{E} \quad ::= \quad [\,] \mid \mathscr{E}\,e \mid v\mathscr{E} \mid [\mathscr{E}]^{e,\sigma} \mid [v]^{\mathscr{E},\sigma} \mid v \lhd^p \mathscr{E} \mid \texttt{let}\ x,y = \mathscr{E}\ \texttt{in}\ e \mid \texttt{case}\ \mathscr{E}\ \texttt{of}\ e_1 \mid e_2$$

The reduction rules for expressions are given in Table 2. Rules [R1–R4] are standard. According to rule [R5], a monitor with a flat contract $\texttt{flat\_c}\ w$ wrapping a value v turns into a busy monitor that checks whether v satisfies the contract by evaluating wv. When the evaluation of wv terminates, the busy monitor reduces to v if the check succeeds (see rule [R6]) or blames p otherwise (see rule [R7]). Note that [R5] *duplicates* v in the reduct. Since endpoints are linear resources, the definition of flat contracts on endpoints (and on linear resources in general) will be forbidden by the type system.

Rules [R8–R14] compute the dual of a contract by reversing the orientation of interactions. For instance, given the contract $\mathrm{C} \stackrel{\text{def}}{=} !(\texttt{flat\_c}\ (\geq 0));?(\texttt{flat\_c}\ (\leq 3));\texttt{end\_c}$ we have

$$\texttt{dual}\ \mathrm{C} \to\to\to ?(\texttt{flat\_c}\ (\geq 0));!(\texttt{flat\_c}\ (\leq 3));\texttt{end\_c}$$

Processes reduce according to the rules in Table 3, where we stack parallel threads instead of separating them by $\parallel$ whenever there is not enough space. Rule [R15] models the initiation of a session through the service channel a. In the reduct, the endpoint b^+ of the new session is returned to the client and the endpoint b^- is passed to the body v of the service in a new thread. The endpoints b^+ and b^- are respectively monitored by the contracts C and $\texttt{dual}\ \mathrm{C}$. The labels in the two monitors reflect the direction of the messages exchanged over the endpoints: in the client, p is negative since p is responsible for the messages sent to the service and q is positive since q is responsible for the messages received by the client; in the spawned thread the responsibilities are reversed. Rule [R16] models the closing of a session.

Rule [R17] models a communication from thread p to thread q on session a, when the endpoints of a are monitored by non-dependent contracts. The rule appears more complicated than it really is. If we erase all the monitors, the rule boils down to

$$\langle \mathscr{E}[\texttt{send}\ v\ a^\iota]\rangle_p \parallel \langle \mathscr{E}'[\texttt{receive}\ a^{\bar\iota}]\rangle_q \to \langle \mathscr{E}[a^\iota]\rangle_p \parallel \langle \mathscr{E}'[(v, a^{\bar\iota})]\rangle_q$$

where we see that the message v just moves from the sender to the receiver. All the additional machinery in [R17] handles contracts. In particular, a^ι in the sender is monitored by a contract of the form $!\mathtt{C}_i\,;\mathtt{D}_i$ and $a^{\overline{\iota}}$ in the receiver is monitored by a contract of the form $?\mathtt{E}_j\,;\mathtt{F}_j$. In the reduct, these contracts must be updated to $\mathtt{D}_i$ and $\mathtt{F}_j$ to reflect the fact that the communication has occurred. Also, the message v is wrapped by all the contracts $\mathtt{C}_i$ and $\mathtt{E}_j$ it is meant to satisfy. Since v is flowing into a^ι, the contracts $\mathtt{C}_i$ are stacked inside out so that the checks on v are performed in the right order. Also, the blame labels are swapped to reflect the correct responsibilities in case contract violations are detected.

Rule [R18] is similar to [R17], except that the two endpoints before the communication are monitored by dependent contracts $!\mathtt{C}_i\,.\,w_{1i}$ and $?\mathtt{D}_j\,.\,w_{2j}$ and the continuation contracts are obtained by applying w_{1i} and w_{2j} to the message. Note that v occurs *several times* in the reduct. As we have discussed for [R5], this could be problematic if v contains linear values such as session endpoints. Hence, the typing of `send_d` and `receive_d` will prevent the definition of contracts that depend on linear values. For simplicity, rules [R17] and [R18] are formulated in such a way that communications occur only when the contracts in the monitors stacked around the involved endpoints are all dependent or all non dependent. This restriction has been introduced solely to avoid considering all the possible combinations of dependent and non-dependent contracts and is relaxed in the implementation, where such distinction is blurred (Section 6.1).

Rules [R19] and [R20] model communications whereby *both* the session type *and* the contract of the endpoints in the reduct (may) depend upon the exchanged message, which is implicitly assumed to be `true` in [R19] and `false` in [R20]. As in [R17] and [R18] the message is checked against a suitable stack of contracts. Then, the receiver injects the continuation endpoint using either `inl` or `inr` correspondingly. Observe that not all of the contracts appearing in the redex also appear in the reduct. In prospect of devising a substructural typing discipline for λCoS, this suggests that contracts should not contain linear resources such as session endpoints.

We stress two facts concerning the reduction rules [R17–R20]. First, we observe that the whole stacks of monitors wrapping the endpoints on which the communication takes place are rearranged in a single reduction step. This is what we mean by "big-step" monitoring semantics, as opposed to the "small-step" monitoring semantics where each monitor is rearranged independently of the others [Findler and Felleisen 2002; Wadler and Findler 2009]. The big-step semantics allows us to rely on some key properties of monitor configurations (Proposition 5.4) when proving the blame soundness results (Section 5). Second, we observe that part of the contracts on the sender side migrate to the receiving side after the communication. For delegations – that is the communication of session endpoints – the migration results in the delegated endpoint being wrapped by further chaperone contracts in addition to those it already has. This is necessary to keep track of the responsibilities of the sender even after the communication has occurred (Example 5.3). For other messages not containing endpoints the migration essentially means that message contracts are always checked on the receiver side, *after* the communication has occurred. In the implementation (Section 6) we consider the alternative "small-step" monitoring semantics where flat contracts for messages are checked on the sender side, *before* communication takes place.

We omit the obvious rules that lift reduction of expression to processes and that close reductions by (process) contexts and structural congruence. In the following we write $\rightarrow^*$ for the reflexive, transitive closure of the $\rightarrow$ relations for expressions and processes. We also introduce some convenient notation for the evaluation of expressions and predicates:

Definition 3.1 (evaluation). We write $e \Downarrow v$ if $e \rightarrow^* v$ and we write $v \in w$ if $wv \Downarrow \mathtt{true}$.

Example 3.2 (function contracts). In this example we show an encoding of function contracts in the style of Findler and Felleisen [2002] as session contracts. Suppose that f is a function and

$\mathtt{C} \mapsto \mathtt{D}$ its contract, where $\mathtt{C}$ and $\mathtt{D}$ are respectively the contracts for the domain and codomain of f. We may represent the function f as a service and the contract $\mathtt{C} \mapsto \mathtt{D}$ as the session contract $!\mathtt{C};?\mathtt{D};\mathtt{end\_c}$, thus:

$$a_f \Leftarrow_p^{!\mathtt{C};?\mathtt{D};\mathtt{end\_c}} \lambda x.\,\mathtt{let}\ y,x = \mathtt{receive}\ x\ \mathtt{in}\ \mathtt{let}\ x = \mathtt{send}\ (f\ y)\ x\ \mathtt{in}\ \mathtt{close}\ x$$

The contract $!\mathtt{C};?\mathtt{D};\mathtt{end\_c}$ gives a natural description of the intended usage protocol for f: the client must send f an argument that satisfies $\mathtt{C}$ and will receive back from f a value that satisfies $\mathtt{D}$. Accordingly, an application $(f\ v)$ can be modeled as follows:

$$\mathtt{let}\ x = \mathtt{connect}\ a_f\ \mathtt{in}\ \mathtt{let}\ x = \mathtt{send}\ v\ x\ \mathtt{in}\ \mathtt{let}\ y,x = \mathtt{receive}\ x\ \mathtt{in}\ \mathtt{close}\ x; \cdots y \cdots$$

Thanks to delegations, the encoding shown here accounts also for higher-order functions and its generalization to dependent functions is straightforward. However, the encoding is purely conceptual and not meant to suggest a practical implementation. Its purpose is to show that the blame soundness results presented in Section 5 are comprehensive enough to also account for the functional part of the calculus, which we have deliberately neglected to keep the formal development as simple and focused on sessions as possible. ■

4 TYPE SYSTEM

We define a session type system for $\lambda\mathtt{CoS}$ loosely inspired to that of Gay and Vasconcelos [2010]. Following Tov and Pucella [2011], we use *kinds* to distinguish *unlimited* types, those denoting values that can be discarded and duplicated, from *linear* types, those denoting values (such as endpoints) that must be used exactly once. The syntax of kinds, types, and session types is given below:

Kind	$\kappa ::= 1 \mid \omega$
Type	$t, s ::= \mathtt{unit} \mid t \times s \mid t + s \mid t \to^{\kappa} s \mid [t] \mid \#T \mid T$
Session Type	$T, S ::= \mathtt{end} \mid ?t.\,T \mid !t.\,T \mid T \,\&\, S \mid T \oplus S$

Types, ranged over by t, s, include base types and standard type constructors. Arrows $\to^{\kappa}$ have a kind annotation κ that indicates whether a function can be applied an arbitrary number of times ($\kappa = \omega$) or must be applied exactly once ($\kappa = 1$). This latter constraint is necessary if the function contains one or more linear values in its closure. We will abbreviate $\to^{\omega}$ with $\to$. The *contract type* $[t]$ describes contracts for values of type t and the *shared channel type* $\#T$ describes channels for initiating sessions with session type T. Session types have the standard constructors for denoting depleted session endpoints ($\mathtt{end}$), input/output operations ($?t.\,T$ and $!t.\,T$), branches ($T \,\&\, S$) and choices ($T \oplus S$).

We now define a relation that assigns kinds to each type. Every type t has kind 1, since using a value of type t exactly once is always safe regardless of t. By contrast, only those types denoting values that can be safely duplicated or discarded have kind ω. Formally:

Definition 4.1 (kinding). We say that t has kind κ if $t :: \kappa$ is derivable by:

$$\mathtt{unit} :: \omega \qquad [t] :: \omega \qquad \#T :: \omega \qquad t \to^{\kappa} s :: \kappa \qquad \frac{t :: \kappa \qquad s :: \kappa}{t \odot s :: \kappa}\ \odot \in \{\times, +\} \qquad \frac{t :: \omega}{t :: 1}$$

Note that contract types and shared channel types have kind ω, whereas the kind of a sum or product depends on that of its components. In particular, a sum/product has kind ω only if both its components do as well. We say that t is *unlimited* if $t :: \omega$ and that t is *linear* otherwise, namely if $t :: \kappa$ implies $\kappa = 1$. For example, $?\mathtt{int}.\mathtt{end}$ and $\mathtt{int} \to^{1} \mathtt{bool}$ and $\mathtt{int} + ?\mathtt{int}.\mathtt{end}$ are all linear, but $?\mathtt{int}.\mathtt{end} \to \mathtt{bool}$ is not.

We introduce *type environments* to keep track of the type of free names in expressions and processes. A type environment Γ is a finite map from names to types written $u_1 : t_1, \ldots, u_n : t_n$. We

Table 4. Type schemes of λCoS constants.

$() : \mathtt{unit}$	$\mathtt{pair} : t \to s \to^{\kappa} t \times s$	$t :: \kappa$
$\mathtt{true} : \mathtt{bool}$	$\mathtt{send} : t \to \mathtt{!}t.T \to^{\kappa} T$	$t :: \kappa$
$\mathtt{false} : \mathtt{bool}$	$\mathtt{receive} : \mathtt{?}t.T \to t \times T$	
$\mathtt{inl} : t \to t + s$	$\mathtt{flat\_c} : (t \to \mathtt{bool}) \to [t]$	$t :: \omega$
$\mathtt{inr} : s \to t + s$	$\mathtt{end\_c} : [\mathtt{end}]$	
$\mathtt{connect} : \#T \to T$	$\mathtt{send\_c} : [t] \to [T] \to [\mathtt{!}t.T]$	
$\mathtt{close} : \mathtt{end} \to \mathtt{unit}$	$\mathtt{receive\_c} : [t] \to [T] \to [\mathtt{?}t.T]$	
$\mathtt{left} : T \oplus S \to T$	$\mathtt{send\_d} : [t] \to (t \to [T]) \to [\mathtt{!}t.T]$	$t :: \omega$
$\mathtt{right} : T \oplus S \to S$	$\mathtt{receive\_d} : [t] \to (t \to [T]) \to [\mathtt{?}t.T]$	$t :: \omega$
$\mathtt{branch} : T \,\&\, S \to T + S$	$\mathtt{choice\_c} : [\mathtt{bool}] \to [T] \to [S] \to [T \oplus S]$	
$\mathtt{dual} : [T] \to [\overline{T}]$	$\mathtt{branch\_c} : [\mathtt{bool}] \to [T] \to [S] \to [T \,\&\, S]$	

write $\emptyset$ for the empty type environment, $\mathrm{dom}(\Gamma)$ for the domain of Γ, namely the set of names for which there is an association in Γ, and Γ, Γ' for the union of Γ and Γ' when $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$. We extend the notion of kinding to type environments and write $\Gamma :: \kappa$ if $\Gamma(u) :: \kappa$ for every $u \in \mathrm{dom}(\Gamma)$.

As usual, we need a way of splitting and combining type environments that is more flexible than disjoint union and, at the same time, prevents the duplication of linear resources. We therefore define a (partial) combination operator + analogous to the one given by Kobayashi et al. [1999]:

Definition 4.2 (environment combination). We write + for the partial operation on type environments such that:

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

Note that $\Gamma + \Gamma'$ is undefined if there are associations $u : t \in \Gamma$ and $u : t' \in \Gamma'$ for the same name u such that either t and t' are different or at least one of them is linear. This prevents multiple uses of a linear resource. Note also that $\Gamma + \Gamma$ is always defined and equal to Γ itself when $\Gamma :: \omega$.

To ensure communication safety, threads are required to perform complementary actions on the peer endpoints of the same session. This is enforced by assigning *dual* session types to peer endpoints. Roughly, the dual of a session type is obtained by swapping inputs with outputs and choices with branches. Formally:

Definition 4.3 (session type duality). The *dual* of T is the session type $\overline{T}$ defined as:

$$\overline{\mathtt{end}} = \mathtt{end} \qquad \overline{\mathtt{?}t.T} = \mathtt{!}t.\overline{T} \qquad \overline{\mathtt{!}t.T} = \mathtt{?}t.\overline{T} \qquad \overline{T \,\&\, S} = \overline{T} \oplus \overline{S} \qquad \overline{T \oplus S} = \overline{T} \,\&\, \overline{S}$$

Note that duality is an involution, that is $\overline{\overline{T}} = T$.

Table 4 shows the type schemes of λCoS constants as associations of the form $\mathtt{c} : t$. In general, each constant may have infinitely many types. The types of data constructors are standard, the only quirk being the second arrow in the type of `pair` which has the same kind as the first element of the pair. This accounts for the possibility that such element has a linear type, in which case exactly one pair must be created to avoid duplicating the element. For example, the partial application $\mathtt{pair}\ a^{\iota}$ is a function which must be applied exactly once in order to preserve the linearity of a^{ι}. The types of the communication primitives are essentially those given by Gay and Vasconcelos [2010] and the types of the contract constructors follow from their semantics. Note that `flat_c`

Table 5. Typing rules.

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

$$\frac{\Gamma :: \omega \qquad \mathtt{c} : t}{\Gamma \vdash \mathtt{c} : t}\ [\textsc{t-const}] \qquad \frac{\Gamma :: \omega}{\Gamma, u : t \vdash u : t}\ [\textsc{t-name}] \qquad \frac{}{\Gamma \vdash \mathtt{blame}\ p : t}\ [\textsc{t-blame}]$$

$$\frac{\Gamma_1 \vdash e : \mathtt{bool} \qquad \Gamma_2 \vdash v : t}{\Gamma_1 + \Gamma_2 \vdash v \lhd^p e : t}\ [\textsc{t-busy-monitor}]$$

$$\frac{\Gamma, x : t \vdash e : s \qquad \Gamma :: \kappa}{\Gamma \vdash \lambda x\,.\,e : t \to^{\kappa} s}\ [\textsc{t-fun}] \qquad \frac{\Gamma_1 \vdash e_1 : t \to^{\kappa} s \qquad \Gamma_2 \vdash e_2 : t}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s}\ [\textsc{t-app}]$$

$$\frac{\Gamma_1 \vdash e_1 : t \qquad \Gamma_2 \vdash e_2 : [t]}{\Gamma_1 + \Gamma_2 \vdash [e_1]^{e_2,p,q} : t}\ [\textsc{t-monitor}]$$

$$\frac{\Gamma_1 \vdash e_1 : t_1 \times t_2 \qquad \Gamma_2, x : t_1, y : t_2 \vdash e_2 : t}{\Gamma_1 + \Gamma_2 \vdash \mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2 : t}\ [\textsc{t-split}]$$

$$\frac{\Gamma_1 \vdash e : t_1 + t_2 \qquad \Gamma_2 \vdash e_i : t_i \to^{\kappa_i} t\ ^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{case}\ e\ \mathtt{of}\ e_1 \mid e_2 : t}\ [\textsc{t-case}]$$

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

$$\frac{\Gamma \vdash e : \mathtt{unit}}{\Gamma \vdash \langle e \rangle_p}\ [\textsc{t-thread}] \qquad \frac{\Gamma_i \vdash P_i\ ^{(i=1,2)}}{\Gamma_1 + \Gamma_2 \vdash P_1 \parallel P_2}\ [\textsc{t-par}]$$

$$\frac{\Gamma, a^+ : T, a^- : \overline{T} \vdash P}{\Gamma \vdash (\nu a)P}\ [\textsc{t-session}] \qquad \frac{\emptyset \vdash \mathtt{c} : [T] \qquad \Gamma \vdash v : \overline{T} \to \mathtt{unit}}{\Gamma + a : \#T \vdash a \Leftarrow^{\mathtt{c}}_p v}\ [\textsc{t-service}]$$

and the dependent contracts `receive_d` and `send_d` require the tested value/message to have an unlimited type, since it may be duplicated.

The typing rules for λCoS (Table 5) are fairly standard, so we only comment on a few notable features. In rules [T-CONST] and [T-NAME] the condition $\Gamma :: \omega$ makes sure that the unused part of type environments does not contain linear resources. Rule [T-FUN] requires the arrow type of a function to have the same kind as the environment in which the function is typed. If there is a linear type in the environment, meaning that the function has a linear value in its closure, then the function can only be applied once. Rule [T-APP] makes use of the + operator for type environments to keep track of the usage of linear resources occurring in the function and its argument. Rules [T-SPLIT] and [T-CASE] are standard. Note once again the use of + for combining type environments. A monitored expression $[e_1]^{e_2,p,q}$ is well typed provided that e_2 is a contract for values of the same type as that of e_1. A busy monitor is well typed provided that the condition being checked has type `bool`. Rule [T-BLAME] states that a blame has any type and is always well typed regardless of the environment. We do not treat blame recovery in the formal model.

Concerning processes, [T-THREAD] and [T-PAR] are as expected. Rule [T-SESSION] introduces a session as a pair of peer endpoints with dual session types. Finally, [T-SERVICE] requires a well-typed service to advertise a contract of the appropriate type and to have a body that accepts one endpoint at each invocation. The body must be unlimited to allow for an arbitrary number of session initiations.

We state subject reduction as a basic sanity check for the type system. Other standard properties – *e.g.* communication safety and session fidelity – also hold, but are irrelevant in this paper.

Theorem 4.4. *If* $\Gamma :: \omega$ *and* $\Gamma \vdash P$ *and* $P \to Q$, *then* $\Gamma \vdash Q$.

5 BLAME SOUNDNESS

In this section we present the soundness of the monitoring mechanism in λCoS, namely the property that in a well-typed program P in which a module p "behaves well", p cannot be blamed. We proceed according to the following roadmap. First, we introduce the notion of *contract entailment* to specify when a contract is "more demanding than" another (Section 5.1). Entailment is a natural generalization of subtyping of session types [Gay and Hole 2005]. Using entailment, we formalize the notion of *locally correct module p* as a module that always honors the contracts of the endpoints it uses. Locality refers to the fact that the correctness of p solely depends on the actions performed by, and on information known to, the module p itself (Section 5.2). Finally, we characterize the *soundness* of a module p as a set of invariant properties of the (busy) monitors in which the label p occurs. A direct consequence of soundness is that a well-typed, locally correct module p cannot be blamed (Section 5.3).

5.1 Contract Entailment

Contract entailment is a relation $\leqslant$ such that, when $\mathrm{C} \leqslant \mathrm{D}$ holds, each value that satisfies C can be used where a value that satisfies D is expected. When C and D are flat contracts, $\leqslant$ boils down to the set-theoretic inclusion between the values that satisfy the respective predicates. For example,

$$\texttt{flat\_c}\ (\geq 3) \leqslant \texttt{flat\_c}\ (\geq 0)$$

since every number greater than or equal to 3 is also greater than or equal to 0. To define entailment when C and D are session contracts, it helps to recall the analogy of contracts as specifications for the messages that can be sent on and received from a session endpoint. In this case, $\mathrm{C} \leqslant \mathrm{D}$ holds if two conditions are satisfied:

(1) Every message that can be received from an endpoint satisfying C can also be received from an endpoint satisfying D.
(2) Every message that can be sent on an endpoint satisfying D can also be sent on an endpoint satisfying C.

Note that C and D occur in different orders according to the direction of exchanged messages. With these intuitions, we formalize entailment below:

Definition 5.1 (contract entailment). We say that e_1 *entails* e_2, written $e_1 \leqslant e_2$, if one of following conditions holds:

(1) $e_1 \Downarrow \texttt{flat\_c}\ w_1$ and $e_2 \Downarrow \texttt{flat\_c}\ w_2$ and $v \in w_1$ implies $v \in w_2$;
(2) $e_1 \Downarrow \texttt{end\_c}$ and $e_2 \Downarrow \texttt{end\_c}$;
(3) $e_1 \Downarrow !\mathrm{C}_1;\mathrm{D}_1$ and $e_2 \Downarrow !\mathrm{C}_2;\mathrm{D}_2$ and $\mathrm{C}_2 \leqslant \mathrm{C}_1$ and $\mathrm{D}_1 \leqslant \mathrm{D}_2$;
(4) $e_1 \Downarrow ?\mathrm{C}_1;\mathrm{D}_1$ and $e_2 \Downarrow ?\mathrm{C}_2;\mathrm{D}_2$ and $\mathrm{C}_1 \leqslant \mathrm{C}_2$ and $\mathrm{D}_1 \leqslant \mathrm{D}_2$;
(5) $e_1 \Downarrow !\texttt{flat\_c}\ v_1 . w_1$ and $e_2 \Downarrow !\texttt{flat\_c}\ v_2 . w_2$ and $v \in v_2$ implies $v \in v_1$ and $w_1 v \leqslant w_2 v$;
(6) $e_1 \Downarrow ?\texttt{flat\_c}\ v_1 . w_1$ and $e_2 \Downarrow ?\texttt{flat\_c}\ v_2 . w_2$ and $v \in v_1$ implies $v \in v_2$ and $w_1 v \leqslant w_2 v$;
(7) $e_1 \Downarrow !\mathrm{C}_1.\mathrm{D}_1:\mathrm{E}_1$ and $e_2 \Downarrow !\mathrm{C}_2.\mathrm{D}_2:\mathrm{E}_2$ and $\mathrm{C}_2 \leqslant \mathrm{C}_1$ and $\mathrm{D}_1 \leqslant \mathrm{D}_2$ and $\mathrm{E}_1 \leqslant \mathrm{E}_2$;
(8) $e_1 \Downarrow ?\mathrm{C}_1.\mathrm{D}_1:\mathrm{E}_1$ and $e_2 \Downarrow ?\mathrm{C}_2.\mathrm{D}_2:\mathrm{E}_2$ and $\mathrm{C}_1 \leqslant \mathrm{C}_2$ and $\mathrm{D}_1 \leqslant \mathrm{D}_2$ and $\mathrm{E}_1 \leqslant \mathrm{E}_2$.

Condition 1 formalizes the set-theoretic inclusion relation between sets of values that satisfy given predicates, whereas condition 2 relates the contract end_c with itself. Conditions 3–4 deal with non-dependent contracts. Entailment is covariant on input prefixes, contravariant on output prefixes, and always covariant on continuation contracts. For example, we have

$$!\texttt{flat\_c}\ (\geq 0);\texttt{end\_c} \leqslant !\texttt{flat\_c}\ (\geq 3);\texttt{end\_c}$$

because the contract on the left-hand side imposes weaker requirements on the messages that can be sent on the endpoint. On the other hand we have

$$?\texttt{flat\_c}\ (\geq 3);\texttt{end\_c} \leqslant ?\texttt{flat\_c}\ (\geq 0);\texttt{end\_c}$$

for the contract on the left-hand side provides stronger guarantees on the messages that can be received from the endpoint.

Conditions 5–6 deal with dependent contracts. The prefix contracts must be flat, since these are the only contracts that can specify constraints on unlimited values. Dependent contracts essentially behave as non-dependent ones, with contravariant outputs and covariant inputs, except that the continuation contracts may depend upon the exchanged message v. For example, we have

$$!\texttt{flat\_c}\ (\geq 0).\lambda x.?\texttt{flat\_c}\ (\leq x);\texttt{end\_c} \leqslant !\texttt{flat\_c}\ (\geq 3).\lambda\_.?\texttt{flat\_c}\ \lambda\_.\texttt{true};\texttt{end\_c}$$

where hereafter we write _ for irrelevant names and sub-terms.

Conditions 7–8 deal with choices and branches. Recalling that a choice is akin to an output and a branch is akin to an input, the definition of entailment essentially follows the same contravariant/covariant pattern we have already seen in the other cases.

Overall, contract entailment is analogous to the subtyping relation for session types [Gay and Hole 2005]. This analogy provides us with the same substitution principle that drives subtyping: when $\mathrm{C} \leqslant \mathrm{D}$, it is safe to use an endpoint with contract C wherever an endpoint with contract D is expected. We will make use of this analogy when defining locally correct modules (Definition 5.2). There are two traits of entailment that set it apart from subtyping. The first one is that entailment is a relation between terms and therefore is undecidable in general. In our setting this is not an issue because the decision as to whether one contract entails another one is not meant to be computed, but is left to the programmer (Section 5.2). The second difference is that entailment is not reflexive in general: according to Definition 5.1, it must be possible to evaluate contracts solely using the reduction relation for expressions, whereas it is possible to write (well-typed) contracts that make use of communication primitives and that reduce only when they occur within a suitable process. In this sense, by embracing Definition 5.1 we take the viewpoint that contracts should be *pure* [Meyer 1992] and their evaluation should not involve side effects. This condition cannot be enforced solely by our type system. From now on, we will make the assumption that $\mathrm{C} \leqslant \mathrm{C}$ holds whenever C is advertised by a service $a \Leftarrow_p^{\mathrm{C}} v$. It is easy to see that entailment is transitive.

5.2 Locally Correct Modules

A *locally correct module* honors the contracts of all the endpoints it uses. Since contracts may depend upon the messages exchanged on such endpoints, we resort to an operational definition of correctness that takes into account all the possible reductions of a process in which label p occurs. The "local" qualification is meant to stress the fact that correctness of a module p only depends on actions performed by, and information known to, the module p itself.

To identify the actions performed by a module p, we introduce *p-contexts* $\mathscr{P}_p$ as terms generated by the grammar

$$\mathscr{P}_p \ ::= \ \langle \mathscr{E} \rangle_p \mid (\mathscr{P}_p \parallel P) \mid (P \parallel \mathscr{P}_p) \mid (\nu a)\mathscr{P}_p$$

where $\mathscr{E}$ is a plain evaluation context. As usual, we write $\mathscr{P}_p[e]$ for the process obtained by filling the hole in $\mathscr{P}_p$ with e. Locally correct modules are defined below.

Definition 5.2 (local correctness). Let $\mathscr{C}$ be the largest relation between blame labels and processes such that $p\ \mathscr{C}\ P$ implies:

(1) $P = \mathscr{P}_p[\texttt{send}\ v\ [\_]^{!\texttt{flat\_c}\ w;\_.\_.\_}]$ implies $v \in w$, and
(2) $P = \mathscr{P}_p[\texttt{send}\ v\ [\_]^{!\texttt{flat\_c}\ w.\_.\_.\_}]$ implies $v \in w$, and

(3) $P = \mathscr{P}_p[\texttt{send}\ [\_]^{\mathrm{C},\multimap,-}\ [\_]^{!\mathrm{D};\multimap,\multimap,-}]$ implies $\mathrm{C} \leqslant \mathrm{D}$, and
(4) $P = \mathscr{P}_p[\texttt{left}\ [\_]^{!\texttt{flat\_c}\ w,\_:\multimap,\multimap,-}]$ implies $\texttt{true} \in w$, and
(5) $P = \mathscr{P}_p[\texttt{right}\ [\_]^{!\texttt{flat\_c}\ w,\_:\multimap,\multimap,-}]$ implies $\texttt{false} \in w$, and
(6) $P \to Q$ implies $p\ \mathscr{C}\ Q$.

We say that module p is *(locally) correct* in P if $p\ \mathscr{C}\ P$ holds.

Condition 1 requires a message v sent over an endpoint monitored by a non-dependent contract $!\texttt{flat\_c}\ w;\_$ to satisfy the predicate w. Condition 2 is similar, except that the contract is a dependent one. Condition 3 concerns delegations whereby an endpoint with contract C is sent as a message that is supposed to satisfy contract D. According to the substitution principle, this is safe if C entails D. Conditions 4–5 concern choices. In these cases, the selected label (either `true` or `false`) is required to satisfy the predicate specified by the choice contract. Finally, condition 6 requires the previous conditions to hold for every possible reduction of P.

Example 5.3. To illustrate the notion of locally correct module, consider the services

$$a \Leftarrow^{\mathrm{C}}_{q} \lambda x_1.\,\texttt{let}\ y_1 = \texttt{connect}\ b\ \texttt{in}\ \texttt{let}\ x_2 = \texttt{send}\ y_1\ x_1\ \texttt{in}\ \texttt{close}\ x_2$$
$$b \Leftarrow^{\mathrm{D}}_{r} \lambda y_1.\,\texttt{let}\ z, y_2 = \texttt{receive}\ y_1\ \texttt{in}\ \texttt{print}\ z^{-1}; \texttt{close}\ y_2$$

with labels q and r and the contracts

$$\mathrm{C} \stackrel{\text{def}}{=} ?\mathrm{E};\texttt{end\_c} \qquad \mathrm{D} \stackrel{\text{def}}{=} !\mathrm{F};\texttt{end\_c} \qquad \mathrm{E} \stackrel{\text{def}}{=} !\texttt{any\_c};\texttt{end\_c} \qquad \mathrm{F} \stackrel{\text{def}}{=} \texttt{flat\_c}\ (\neq 0)$$
$$\overline{\mathrm{C}} \stackrel{\text{def}}{=} !\mathrm{E};\texttt{end\_c} \qquad \overline{\mathrm{D}} \stackrel{\text{def}}{=} ?\mathrm{F};\texttt{end\_c}$$

noting that C and D are the contracts advertised by q and r, respectively. Also note that $\texttt{dual}\ \mathrm{C} \Downarrow \overline{\mathrm{C}}$ and $\texttt{dual}\ \mathrm{D} \Downarrow \overline{\mathrm{D}}$. Service r receives a non-zero number z and prints its inverse. Service q initiates a session with r and then delegates the corresponding endpoint y_1 to its client. Consider also the p-labeled client

$$\langle \texttt{let}\ x_1 = \texttt{connect}\ a\ \texttt{in}\ \texttt{let}\ y_1, x_2 = \texttt{receive}\ x_1\ \texttt{in}\ \texttt{let}\ y_2 = \texttt{send}\ 0\ y_1\ \texttt{in}\ \texttt{close}\ x_2; \texttt{close}\ y_2 \rangle_p$$

which connects to service q, receives a session endpoint y_1 from it, and then sends 0 on y_1. Observe that p connects to a service whose contract is $?(!\texttt{any\_c};\texttt{end\_c});\texttt{end\_c}$. Hence, p believes that the contract of y_1 is $!\texttt{any\_c};\texttt{end\_c}$ and that it is allowed to send 0 on y_1.

Working out the reductions of the client in parallel with the two services we obtain the following assignments for the variables occurring in modules p, q, and r:

$$p \begin{bmatrix} x_1 = [c^+]^{\mathrm{C},q,p} \\ y_1 = [[[d^+]^{\mathrm{D},r,q}]^{\mathrm{E},q,p}]^{\mathrm{E},q,p} \\ x_2 = [c^+]^{\texttt{end\_c},q,p} \\ y_2 = [[[d^+]^{\texttt{end\_c},r,q}]^{\texttt{end\_c},q,p}]^{\texttt{end\_c},q,p} \end{bmatrix} \qquad q \begin{bmatrix} x_1 = [c^-]^{\overline{\mathrm{C}},p,q} \\ y_1 = [d^+]^{\mathrm{D},r,q} \\ x_2 = [c^-]^{\texttt{end\_c},p,q} \end{bmatrix}$$

$$r \begin{bmatrix} y_1 = [d^-]^{\overline{\mathrm{D}},q,r} \\ z \approx [[[[0]^{\texttt{any\_c},p,q}]^{\texttt{any\_c},p,q}]^{\mathrm{F},q,r}]^{\mathrm{F},q,r} \\ y_2 \approx [d^-]^{\texttt{end\_c},q,r} \end{bmatrix}$$

An assignment $x = v$ denotes the fact that the program reduces to a state in which the substitution $\{v/x\}$ is actually performed in the respective thread. The assignment $z \approx e$ in thread r does not materialize into an actual substitution because 0 does not satisfy F and the evaluation of e eventually blames q. A similar observation holds for y_2.

According to Definition 5.2, in order to identify the locally correct modules we have to consider all the output operations performed by these threads. We deduce that r is trivially locally correct, since r does not perform any output operation. Concerning p, we see that it eventually performs one output operation $\texttt{send}\ 0\ [[[d^+]^{\mathrm{D},r,q}]^{\mathrm{E},q,p}]^{\mathrm{E},q,p}$ where $\mathrm{E} = \texttt{!any\_c;end\_c}$. Since $0 \in \texttt{any\_c}$, we deduce that p is also locally correct. Notice that, in order to decide whether p honors the contract of d^+, we only look at the topmost monitor wrapping d^+ which, as we will see, reflects the local knowledge of p concerning this endpoint. Concerning q, we observe that it eventually performs the delegation $\texttt{send}\ [d^+]^{\mathrm{D},r,q}\ [c^-]^{\overline{\mathrm{C}},p,q}$ where $\overline{\mathrm{C}} = \texttt{!(!any\_c;end\_c);end\_c}$. In particular, the message being sent is an endpoint with contract $\mathrm{D} = \texttt{!flat\_c}\ (\neq 0)\texttt{;end\_c}$ whereas q is supposed to send an endpoint with contract $\texttt{!any\_c;end\_c}$. Observe that $\texttt{!flat\_c}\ (\neq 0)\texttt{;end\_c} \not\leqslant \texttt{!any\_c;end\_c}$ because $\texttt{any\_c} \not\leqslant \texttt{flat\_c}\ (\neq 0)$ and entailment is contravariant with respect to outputs. That is, q violates item 3 of Definition 5.2 and therefore it is not locally correct. ■

Given that the conditions in Definition 5.2 are stated by actually running the process P in which label p occurs, one could wonder whether their enforcement demands unreasonable skills (such as divination or omniscience) to the programmer writing the code of p. The "local" qualification in Definition 5.2 is meant to indicate that this is not the case, namely that the definition provides effective guidance for writing correct code. We make two observations in this respect. First, all the conditions of Definition 5.2 only concern actions that are actually *performed* by module p, not by other modules. We concede that p may receive code from other modules (functions can be sent as messages) and, if this happens, Definition 5.2 assumes such code to behave well too. This assumption could be relaxed using a finer mechanism for associating blame labels with code [Findler and Felleisen 2002], but we leave this refinement for future work. The second observation is that the contracts $\texttt{flat\_c}\ w$, C, and D mentioned in conditions 1–5 are always found in the *topmost* monitors wrapping messages and endpoints. Inspection of the rules in Table 3 reveals that these contracts are always (continuations of) those advertised by a service previously invoked by p. We can formalize this claim by showing that the negative label of every topmost monitor occurring in module p is p itself. Given that labels do not migrate between monitors, this means that the contract of the topmost monitor wrapping an endpoint in module p faithfully reflects the knowledge of p about that endpoint.

Proposition 5.4. *If P is a user process such that $P \to^* \mathscr{P}_p[\texttt{c}\ v_1 \cdots v_n\ [\_]^{\rightarrowtail,\rightarrowtail,q}]$ where $\texttt{c} \in \{\texttt{send}, \texttt{left}, \texttt{right}\}$, then $q = p$.*

In conclusion, the programmers of module p have all the information to consciously write locally correct code if they know the contracts of the services invoked by p. This is the weakest requirement we could reasonably ask.

Example 5.5. Consider again the processes of Example 5.3. The programmer of module p statically knows that the module is establishing a session with the service a whose contract is $\mathrm{C} = \texttt{?(!any\_c;end\_c);end\_c}$. According to this contract, the programmer assumes that the endpoint y_1 received from a can be used for sending *any* message, as specified by the contract any_c, even though we know, by looking at the assignments determined in Example 5.3, that the innermost monitor wrapping y_1 specifies a stricter constraint. From the same assignments we also see that C will indeed be the contract found in the topmost monitor wrapping y_1. This means that the assumptions made by the programmer concerning y_1 correspond to the conditions that determine the local correctness of the module p.

The programmer of module q statically knows that the module is establishing a session y_1 with the service b whose contract is $\mathrm{D} = \texttt{!(flat\_c}\ (\neq 0)\texttt{);end\_c}$. The same programmer is also aware that module q is itself a service advertizing the contract C, which requires the service to send on x_1

an endpoint with contract $\text{E} = \,!\texttt{any\_c};\texttt{end\_c}$. The mistake made by the programmer is detectable by comparing D (the contract of the endpoint y_1 that the service actually sends on x_1) and E (the contract of the endpoint that the service is supposed to send on x_1) and noticing that $\text{D} \not\leqslant \text{E}$. ■

5.3 Local Correctness Implies Blame Soundness

A locally correct module p cannot be blamed. The key insight for proving this result is that *local* correctness of p in a process P grants a number of *global* properties on the (busy) monitors referring to p in P. Crucially, these properties hold *anywhere* in P (not just within module p) and are *invariant* under arbitrary reductions of P.

Henceforth, we write $P \supset e$ (or equivalently $e \subset P$) if the sub-expression e occurs in P.

Definition 5.6 (module soundness). We say that module p is *sound* in P if:

(1) $P \supset [v]^{\texttt{flat\_c}\ w,p,-}$ implies $v \in w$;
(2) $P \supset v \triangleleft^{p} e$ implies $e \Downarrow \texttt{true}$;
(3) $P \supset [v \triangleleft\!\text{-}\ e]^{\texttt{flat\_c}\ w,p,-}$ and $e \Downarrow \texttt{true}$ imply $v \in w$;
(4) $P \supset [[\_]^{\text{C},-,q}]^{\text{D},r,-}$ and $p \in \{q, r\}$ imply $\text{C} \leqslant \text{D}$.

Conditions 1–3 state that all values v wrapped by a (busy) monitor where p is deemed responsible satisfy the contract/condition in the monitor. Condition 4 concerns stacked monitors where p is either the positive or negative label depending on whether p occurs in the outermost or innermost monitor. These configurations roughly correspond to *casts* [Wadler and Findler 2009], whereby a value that is supposed to satisfy some contract C is used in a place where a value that satisfies D is expected. Regardless of whether p is the provider ($p = r$) or consumer ($p = q$) of such value, the entailment $\text{C} \leqslant \text{D}$ protects p in the sense that, if p risks of being blamed, another module will be blamed beforehand.

The key lemma states that soundness of p is preserved by reductions of P, provided that p is locally correct in P.

Lemma 5.7 (soundness preservation). *If $\Gamma \vdash P$ where $\Gamma :: \omega$ and p is locally correct and sound in P and $P \to Q$, then p is sound also in Q.*

Blame soundness is an easy corollary of Lemma 5.7. Note that p is trivially sound in every user process, since the clauses of Definition 5.6 solely concern the runtime syntax.

Theorem 5.8 (blame soundness). *If $\Gamma \vdash P$ where P is a user process and p is locally correct in P, then $P \to^{*} Q$ implies $\texttt{blame}\ p \notin Q$.*

It is worth comparing Theorem 5.8 with similar results in the literature. The concepts most closely related to blame soundness are *blame correctness* [Dimoulas et al. 2011, 2012] and *blame safety* [Wadler 2015; Wadler and Findler 2009].

Blame correctness is a general property of a contract system guaranteeing that, whenever a module p may be blamed for a contract violation, it is because a value owned by (*i.e.* generating from) p is being checked against a flat contract which p was supposed to honor. In the literature, this notion has arisen following the observation that alternative monitoring strategies may yield different blame assignments [Blume and McAllester 2006]. In contrast, blame soundness is a property of a particular module p and is formulated as the logical transposition of blame correctness: it states that, if p is never responsible for a contract violation, then p will never be blamed. This difference between blame correctness and blame soundness is reflected also in the techniques used for proving the two results. Blame correctness is proved by tracking the ownership of values and the obligations of modules with respect to contracts. In particular, there is no concept akin to that of "correct" module. The proof of blame soundness rests on a characterization of those modules that do not

violate contracts and is closer in style to a type safety result, where Lemma 5.7 plays the role of subject reduction.

The blame calculus [Wadler 2015; Wadler and Findler 2009] is a model of programs comprising both more-typed and less-typed modules. The interaction between modules adopting different typing disciplines is governed by type casts that are checked at runtime and may trigger blames. Blame safety is the property guaranteeing that, in case a blame is triggered, it always concerns a less-typed module. A locally correct module can be seen as a software component providing stronger guarantees about its behavior than those granted by the type system alone. In this sense, blame soundness resembles blame safety in that it guarantees that blames always concern modules that provide weaker guarantees about their correctness. Unlike well typing, local correctness is formulated operationally and not through a set of typing rules.

6 OCaml IMPLEMENTATION

In Sections 3–5 we have presented the model of λCoS and studied its metatheory. With this formal background, we now present a practical implementation of λCoS communication primitives and session monitoring in OCaml. There are a few differences between the model and the implementation. Some concern the operational semantics and are suggested by practical considerations (Section 6.1). Others concern OCaml's type system, whose support for parametric polymorphism and recursive types allows us to devise a more expressive API to our library of monitored sessions (Section 6.2). We conclude the section with a final example that illustrates all these features at work (Section 6.3).

6.1 Alternative Monitoring Semantics

According to the reduction rule [R15], when a session is initiated with a service $a \Leftarrow^{\mathrm{C}}_{p} v$, the client endpoint is monitored by the contract C advertised by the service and the service endpoint is monitored by the dual contract computed by `dual` C. We dub this semantics, where each peer of a session has its own monitor, *double-sided monitoring* in contrast to *single-sided monitoring* where only one peer is wrapped. Having monitors on both peers assures the presence of at least one chaperone contract for each endpoint in the system. To see the reason why this is technically convenient, consider item 3 of Definition 5.2, which concerns delegations: the guarantee that there is a monitor wrapping both the message and the endpoint on which the message is sent is a direct consequence of double-sided monitoring (*cf.* rule [R15]) and allows us to easily refer to the contracts C and D of the involved endpoints. Without this guarantee we would have four versions of item 3, depending on whether each endpoint is monitored or not. Even worse, we would have to *compute* the dual contract of each lone endpoint from that wrapping its peer, which could be located anywhere in the system. With the guarantee that each endpoint is monitored, Definition 5.2 remains reasonably compact and need not refer to parts of the system other than module p.

In practice, double-sided monitoring induces a useless overhead because each message exchanged through the peer endpoints of a session is checked against the same contract twice, once for each peer. This is a consequence of the fact that `dual` only inverts the direction of the exchanged messages without affecting their contracts (rules [R8–R14]). The overhead caused by double-sided monitoring is clearly illustrated by Example 5.3, where several values end up being wrapped by adjacent monitors having the same contract and the same labels. We can eliminate this overhead by adopting single-sided monitoring, where session initialization attaches just one monitor to the client endpoint. From an operational standpoint, this amounts to replacing rule [R15] with

$$[\text{R21}] \qquad \langle \mathscr{E}[\texttt{connect}\ a] \rangle_p \parallel a \Leftarrow^{\mathrm{C}}_{q} v \mapsto (\nu b)(\langle \mathscr{E}[[b^+]^{\mathrm{C},q,p}] \rangle_p \parallel \langle v\ b^- \rangle_q) \parallel a \Leftarrow^{\mathrm{C}}_{q} v$$

that leaves b^- unmonitored. While it is obvious that by removing one monitor we reduce the points in the program that can generate blame, it is not entirely obvious that no new blame is introduced. The next result guarantees that this is indeed the case:

Proposition 6.1. *Let $\mapsto$ be the relation defined by the rules in Tables 2 and 3 except that* [R15] *is replaced by* [R21]. *If P is a user process and $P \mapsto^* \supset \texttt{blame}\ p$, then $P \to^* \supset \texttt{blame}\ p$.*

In fact we conjecture that the converse of Proposition 6.1 also holds, namely that single-sided and double-sided monitoring yield the same blames. Proving this fact turns out not to be trivial as the exact relationship between processes resulting from the two monitoring semantics is hard to formalize. As it stands, Proposition 6.1 is enough to conclude that single-sided monitoring does not affect the blame soundness result (Theorem 5.8).

We now turn the attention to the rules [R17–R20], which are defined in such a way that a single reduction step rearranges a whole stack of monitors. This formulation of the operational semantics – which we dub *big-step monitoring* – is convenient to prove various auxiliary results, including Proposition 5.4 and Lemma 5.7, which are key ingredients of blame soundness (Theorem 5.8). The issue with big-step monitoring is that the contracts concerning a communicated message migrate along with the message from the sender to the receiver. Given that contracts are ordinary expressions, this migration is, in effect, a form of code mobility which may pose problems, especially in a distributed system. In addition, the message is not checked against the contracts on the sender side until the communication has completed. This means that contract violations might go undetected if, for instance, a deadlock or livelock prevents the communication from occurring.

For these reasons, it makes sense to consider a *small-step monitoring* semantics that deals with monitors one at a time and that is closer in style to the semantics given by Findler and Felleisen [2002]. We illustrate small-step monitoring for a communication across endpoints monitored by non-dependent contracts. Communication goes through two distinct phases. During the first phase, the monitors on the endpoints are disassembled and distributed on the message and around the whole applications involving `send` and `receive`. Formally:

$$\text{[R22]}\quad \texttt{send}\ v\ [\varepsilon]^{!\mathrm{C};\mathrm{D},\sigma} \rightsquigarrow [\texttt{send}\ [v]^{\mathrm{C},\neg\sigma}\ \varepsilon]^{\mathrm{D},\sigma}$$

$$\text{[R23]}\quad \texttt{receive}\ [\varepsilon]^{?\mathrm{C};\mathrm{D},\sigma} \rightsquigarrow \texttt{let}\ x,y = \texttt{receive}\ \varepsilon\ \texttt{in}\ ([x]^{\mathrm{C},\sigma},[y]^{\mathrm{D},\sigma})$$

Once all the contracts have been stripped off and suitably rearranged/checked, the actual communication may take place:

$$\text{[R24]}\quad \langle \mathscr{E}[\texttt{send}\ v\ a^{\iota}] \rangle_p \parallel \langle \mathscr{E}'[\texttt{receive}\ a^{\bar{\iota}}] \rangle_q \rightsquigarrow \langle \mathscr{E}[a^{\iota}] \rangle_p \parallel \langle \mathscr{E}'[(v, a^{\bar{\iota}})] \rangle_q$$

With small-step monitoring, contract migration only happens during the communication of endpoints (delegations). Arguably, the contexts in which this form of communication is allowed also allow for the migration of contracts. In addition, rule [R22] makes sure that messages different from endpoints are checked against contracts *before* the `send` can reduce further because $[v]^{\mathrm{C},\neg\sigma}$ is not a value when C is a flat contract. This means that these checks are performed even if the communication does not occur and their cost is charged to the sender. We illustrate these differences between big-step and small-step monitoring in the following example.

Example 6.2. Let P be the parallel composition

$$\begin{array}{l} \langle \texttt{let}\ x = \texttt{send}\ 0\ [a^+]^{!\texttt{flat\_c}\ (\neq 0);\texttt{end\_c},q,p}\ \texttt{in} \\ \ \texttt{let}\ y = \texttt{send}\ 1\ [b^+]^{!\texttt{any\_c};\texttt{end\_c},q,p}\ \texttt{in} \\ \ \texttt{close}\ x; \texttt{close}\ y \rangle_p \end{array} \quad \parallel \quad \begin{array}{l} \langle \texttt{let}\ \_,y = \texttt{receive}\ [b^-]^{?\texttt{any\_c};\texttt{end\_c},p,q}\ \texttt{in} \\ \ \texttt{let}\ \_,x = \texttt{receive}\ [a^-]^{?\texttt{flat\_c}\ (\neq 0);\texttt{end\_c},p,q}\ \texttt{in} \\ \ \texttt{close}\ x; \texttt{close}\ y \rangle_q \end{array}$$

where p is sending two messages (0 and 1) and q is waiting for them. Notice that p is sending a message on a^+ first and then on b^+, whereas q is waiting for a message from b^- first and then

a^-. Because of the different order of communications (and of the synchronous communication model used in λCoS), the process P is stuck according to the big-step monitoring semantics. That is, $P \not\twoheadrightarrow$. Notice also that p is violating the contract of a^+, which requires the message to be a number different from 0. Because of the deadlock, this violation is not detected by the big-step monitoring semantics. On the contrary, under small-step monitoring the p thread reduces thus

$$
\begin{array}{ll}
\langle \texttt{let}\ x = \texttt{send}\ 0\ [a^+]^{!\texttt{flat\_c}\ (\neq 0);\texttt{end\_c},q,p}\ \texttt{in}\ \cdots\rangle_p & \\
\quad \rightsquigarrow \langle \texttt{let}\ x = [\texttt{send}\ [0]^{\texttt{flat\_c}\ (\neq 0),p,q}\ a^+]^{\texttt{end\_c},q,p}\ \texttt{in}\ \cdots\rangle_p & \text{by [R22]} \\
\quad \rightarrow \langle \texttt{let}\ x = [\texttt{send}\ (0 \triangleleft^p\ (\neq)\ 0\ 0)\ a^+]^{\texttt{end\_c},q,p}\ \texttt{in}\ \cdots\rangle_p & \text{by [R5]} \\
\quad \rightarrow \langle \texttt{let}\ x = [\texttt{send}\ (0 \triangleleft^p\ \texttt{false})\ a^+]^{\texttt{end\_c},q,p}\ \texttt{in}\ \cdots\rangle_p & \text{semantics of} \neq \\
\quad \rightarrow \langle \texttt{let}\ x = [\texttt{send}\ (\texttt{blame}\ p)\ a^+]^{\texttt{end\_c},q,p}\ \texttt{in}\ \cdots\rangle_p & \text{by [R7]}
\end{array}
$$

yielding a blame for p even if the communication does not occur. ■

It is possible to prove a (limited) form of equivalence between big-step and small-step monitoring by considering those configurations where a communication is about to occur:

Proposition 6.3. *Let $\rightsquigarrow$ be the relation defined by the rules in Tables 2 and 3 except that rule* [R17] *is replaced by* [R22–R24]. *If*

$$
\begin{array}{rcl}
P & \stackrel{\text{def}}{=} & \langle \mathscr{E}[\texttt{send}\ v\ [a^\iota]^{\overrightarrow{!C;D,\sigma}}]\rangle_p \parallel \langle \mathscr{E}'[\texttt{receive}[a^{\bar\iota}]^{\overrightarrow{?E;F,\varrho}}]\rangle_q \\
Q_1 & \stackrel{\text{def}}{=} & \langle \mathscr{E}[[a^\iota]^{\overrightarrow{D,\sigma}}]\rangle_p \parallel \langle \mathscr{E}'[(v, [a^{\bar\iota}]^{\overrightarrow{F,\varrho}})]\rangle_q \\
Q_2 & \stackrel{\text{def}}{=} & \langle \mathscr{E}[[a^\iota]^{\overrightarrow{D,\sigma}}]\rangle_p \parallel \langle \mathscr{E}'[([[v]^{\overleftarrow{C,\neg\sigma}}]^{\overrightarrow{E,\varrho}}, [a^{\bar\iota}]^{\overrightarrow{F,\varrho}})]\rangle_q
\end{array}
$$

then either

(1) $P \rightsquigarrow^* Q$ *and* $P \rightarrow^* Q$ *for some* $Q \in \{Q_1, Q_2\}$, *or*
(2) $P \rightsquigarrow^* \supset \texttt{blame}\ r$ *and* $Q \rightarrow^* \supset \texttt{blame}\ r$ *for some* r.

In words, regardless of the monitoring semantics and starting from a configuration involving a sender and a receiver, either the reduction converges to the same state Q_i in which the message has been successfully delivered to the destination, or the same module r is blamed by both semantics. There are two possible resulting states Q_1 and Q_2 depending on whether v is a ground value or a (monitored) endpoint. In the first case, the monitors around v turn into busy monitors and eventually disappear. In the second case, they accumulate around v. Analogous results can be proven for all the other configurations involving communication primitives.

We also note that the small-step semantics provides for greater flexibility in that it allows the reduction of configurations where stacked monitors have a mixture of dependent and non-dependent contracts because each monitor is handled individually.

Despite these differences between big-step and small-step monitoring, the given criterion of local correctness and the blame soundness result are also relevant for our implementation. After all, what small-step monitoring does is to introduce a number of intermediate steps in which the relationships between adjacent monitors are (temporarily) lost and where the actual knowledge of the programmer with respect to the contract of the used endpoints is not as easy to characterize as in Proposition 5.4. In case communications are guaranteed to occur, Proposition 6.3 shows that both monitoring semantics either yield the same blame or eventually converge to the same reduct. A difference remains in that the thread from which the blame originates may not be the same: in the big-step monitoring semantics it is always the receiver of a message that possibly detects a violation (*cf.* thread r in Example 5.3), whereas in the small-step monitoring semantics checks are in general performed on both sides of the session and the sender may yield blames too (*cf.* thread p in Example 6.2).

6.2 Monitoring Sessions in OCaml

In this section we illustrate the key aspects of an OCaml module that implements λCoS communication primitives. Instead of building the primitives from scratch, we obtain them as wrappers of the corresponding primitives provided by FuSe, an OCaml library of binary sessions [Padovani 2017]. This way we do not have to delve into low-level details concerning the encoding of session types or the implementation of the session primitives and instead we can focus on the aspects strictly related to contract monitoring. The fact that we can build λCoS primitives on top of FuSe is a sign that our monitoring technique is modular and should be portable to other session libraries for possibly different programming languages.

Even though FuSe has its own OCaml representation of session types, we keep using the metavariables T and S to improve readability. In particular, we write T `st` for the OCaml type that denotes a lone (*i.e.*, unmonitored) FuSe endpoint with session type T. Similarly, we write T `mt` for the OCaml type that denotes a possibly monitored endpoint. OCaml's support for parametric polymorphism in conjunction to FuSe's representation of session types makes it possible to also represent session type variables standing for unknown session types. Hereafter we use A, B to range over session type variables and A `st`, A `mt` for their corresponding representations in FuSe and our λCoS implementation. The FuSe representation of session types makes it easy to switch from a session type to its dual [Padovani 2017]. We will write $\overline{A}$ `st` to refer to the dual of A `st`.

Concerning the session communication primitives, we use the prefix FuSe to disambiguate them from those we are going to implement. As an example,

```
FuSe.send : α → !α.A st → A st
```

is the signature of the send primitive provided by FuSe, which we will use for providing a suitable wrapper

```
send : α → !α.A mt → A mt
```

Following Hinze et al. [2006], we represent the contract forms introduced in Section 3 using the constructors of a generalized algebraic data type (GADT):

```
1  type [_] =
2    | Flat    : (α → bool) → [α]
3    | End     : [end]
4    | Receive : [α] * (α → [A mt]) → [?α.A mt]
5    | Send    : [α] * (α → [A mt]) → [!α.A mt]
6    | Branch  : [bool] * [A mt] * [B mt] → [A & B mt]
7    | Choice  : [bool] * [A mt] * [B mt] → [A ⊕ B mt]
```

According to the OCaml syntax for GADTs, the type parameter of the data type is left unspecified (see _ on line 1) and each constructor is explicitly annotated with its type (lines 2–7). These contract constructors are in close correspondence with those of Table 1 and are kept private in our OCaml module for λCoS which instead exports one (curried) function for each of them. For example, `flat_c` is defined thus

```
let flat_c w = Flat w
```

In OCaml the function `flat_c` is polymorphic and has type `(α → bool) → [α]`, hence it can be used to define flat contracts on values of any type. The typical example is `any_c : [α]` used in several occasions and defined below:

```
let any_c = Flat (fun _ → true)
```

Unlike λCoS, OCaml's type system is not substructural and there is no distinction between linear and unlimited types. Consequently, we cannot statically ensure that flat contracts are used only for unlimited values. As we will see shortly, violations on the usage of flat contracts will be detected dynamically using a runtime check.

The remaining costructors are in correspondence with the contract forms for session endpoints. In particular, End describes an endpoint that can only be closed. We have:

```
let end_c = End
```

Send is the constructor for a dependent output contract, which is built up from a contract [α] for the communicated value and a function (α → [A mt]) that generates the continuation contract. The corresponding combinator send_d is defined thus:

```
let send_d k1 k2 = Send (k1, k2)
```

The lack of distinction between linear and unlimited types in OCaml allows us to define non-dependent contract combinators as particular instances of the corresponding dependent version. Thus, we have

```
let send_c k1 k2 = send_d k1 (fun _ → k2)
```

The primitives receive_c, receive_d, choice_c, and branch_c are defined similarly. As pointed out by a reviewer, it is also sensible to provide specializations of choice_c and branch_c that mandate the selection of a particular branch. For instance

```
let left_choice_c k = choice_c (flat_c id) k any_c
let right_choice_c k = choice_c (flat_c not) any_c k
```

can be used to specify that the user of an endpoint is required to select the left (respectively, right) branch of a choice. Indeed, the (flat_c id) contract is satisfied by the true label whereas the (flat_c not) contract is satisfied by the false label.

It is now time to provide the concrete representation of a λCoS endpoints as possibly monitored FuSe endpoints. In general we have to take into account the possibility that an endpoint is wrapped by an arbitrary number of monitors, each carrying a chaperone contract. For this reason, we use a recursive data type:

```
type A mt =
  | Channel of linearity_tag_type * A st
  | Monitor of [A mt] * string * string * A mt
```

Note that this definition and the GADT for contracts are mutually dependent and must be introduced jointly in OCaml. Here we have kept them separate for the sake of a more gradual presentation. The Channel constructor represents a lone endpoint (*cf.* Table 1) and consists of a FuSe endpoint and a linearity tag, which will be used for the runtime detection of linearity violations on flat contracts. The definition of linearity_tag_type is uninteresting. We only remark that it is a singleton type whose unique inhabitant is kept private to the module and can only occur in a Channel constructor. The Monitor constructor represents a monitored endpoint, consisting of a contract (of type [A mt]), two blame labels (of type string), and an underlying, possibly monitored endpoint of type A mt.

We now illustrate in detail the implementation of send, the other communication primitives are analogous. Recall that send has two arguments, a message v and an endpoint ep on which v must be transmitted, and returns the same endpoint (*cf.* Table 3). Keeping this in mind, we have:

```
1  let rec send v =
2    function
3    | Channel (lin, ep) → Channel (lin, FuSe.send v ep)
4    | Monitor (Send (k, w), pos, neg, ep) →
5       wrap (w v) pos neg (send (wrap k neg pos v) ep)
6    | Monitor (Flat _, _, _, _) → assert false (* IMPOSSIBLE *)
```

The case when `ep` is a lone endpoint (line 3) corresponds to the reduction rule [R24] and consists in invoking the corresponding `FuSe` primitive and turning the resulting `FuSe` endpoint into a λ`CoS` endpoint through the `Channel` constructor. The case when `ep` is a monitored endpoint (lines 4–5) corresponds to the reduction rule [R22]. In this case, the contract is disassembled into a contract `k` for the message being transmitted and a function `w` that is applied to the message to compute the continuation contract. The auxiliary function `wrap` (described shortly) takes care of checking that contracts are satisfied or attaches chaperone contracts to session endpoints. Note the swapping of blame labels in the inner application of `wrap` and the recursive application of `send`, which takes care of others chaperone contracts in `ep` and eventually performs the communication. The last case in the definition of `send` (line 6) never applies and is only meant to prevent complaints about a non-exhaustive pattern matching. The point is that when `send` is applied to a monitored endpoint `Monitor (k, _, _, ep)` the type of `send` ensures that `k` has type `[!`α`.`A `mt]` and the only constructors whose type is unifiable with this one are `Send` and `Flat`, which applies to any value but is never attached as a chaperone contract by `wrap`.

The last key ingredient of the implementation is indeed `wrap`, whose purpose is to check whether a value satisfies a given contract and to blame the guilty party if this is not the case. The structure of `wrap`, shown below, is analogous to that of the homonymous function defined by Findler and Felleisen [2002], the main difference being that, in our case, `wrap` deals with contracts concerning session endpoints.

```
1  let wrap : type a. a ct → string → string → a → a = fun k pos neg v →
2    match k with
3    | Flat w         → if unlimited v && w v then v else raise (Blame pos)
4    | End as k       → Monitor (k, pos, neg, v)
5    | Receive _ as k → Monitor (k, pos, neg, v)
6    | Send _ as k    → Monitor (k, pos, neg, v)
7    | Branch _ as k  → Monitor (k, pos, neg, v)
8    | Choice _ as k  → Monitor (k, pos, neg, v)
```

An application `wrap k pos neg v` inspects the structure of the contract `k` (line 2). When `k` is a flat contract the message is verified to be unlimited and to satisfy the predicate `w` (line 3). If both conditions are satisfied, the value `v` is returned. On the contrary, the sender is blamed for a contract violation. The `unlimited` auxiliary function is implemented by checking that (the runtime representation of) `v` does not contain any occurrence of `linear_tag`. Overall, this case in the definition of `wrap` corresponds to the rules [R5–R7] in Table 2. In the remaining cases (lines 4–8), `wrap` cannot check immediately whether the endpoint satisfies the contract `k`. Therefore, the endpoint is wrapped by a monitor with the contract `k` and the blame labels `pos` and `neg`. Two remarks about the typing of `wrap` are in order. First, it is necessary to provide a type annotation to `wrap` (line 1) to inform `OCaml` that the type parameter α is *locally abstract* (the explicit quantification `type` α means this). This way, `OCaml` can refine this type parameter depending on pattern matching [Garrigue and Normand 2011]. Second, the lines 5–9 cannot be collapsed into a single case, even if they all

bind just one variable k and the right hand side of each pattern matching rule is behaviorally the same, for the type of k is different in each case.

6.3 Extended Example: List Forwarding

In this section we present a final example that illustrates all the features of the OCaml implementation of λCoS, including dependent contracts. We also take advantage of OCaml's support for parametric polymorphism and recursive types, which we have omitted in the formal model of λCoS for the sake of simplicity.

The following function

```
1  let forwarder_body x =
2    let rec aux y =
3      function
4      | []     → close (right y)
5      | v :: l → aux (send v (left y)) l in
6    let l, x = receive x in (* receive the elements to be forwarded *)
7    let y, x = receive x in (* receive the destination endpoint     *)
8    close x; aux y l
```

models the body of a forwarding service that delivers a list of elements to a given recipient, one element at a time. The service interacts with the client using the endpoint x, from which it receives a list l of elements (line 6) and another endpoint y on which the elements of the list should be forwarded (line 7). The main loop of the service (lines 2–5) iterates over the list: when the list is empty (line 4), the service selects the "right" branch of the protocol and closes the endpoint; when the list contains at least one element v, the service selects the "left" branch of the protocol, it sends v on y, and then iterates over the tail of the list (line 5).

OCaml infers for forwarder_body the signature

```
val forwarder_body : ?(α list).?(rec A.(!α.A) ⊕ end).end mt → unit
```

where rec A.(!α.A) ⊕ end stands for the equi-recursive session type T that satisfies the equation T = (!α.T) ⊕ end.[1] As the type suggests, forwarder_body performs an arbitrary number of outputs on the delegated endpoint y, and by parametricity we also deduce that the elements being sent in these outputs must come from the list l. However, we do not know whether the number of forwarded elements actually matches the length of the list. The service can advertise this guarantee by means of the following contract:

```
1  let forwarder_c =
2    let rec fwd n =
3      if n > 0 then (* more list elements *)
4        left_choice_c (send_c any_c (fwd (n - 1)))
5      else (* no more list elements *)
6        right_choice_c end_c
7    in send_d any_c (fun l → send_c (fwd (List.length l)) end_c)
```

The contract starts with the specification of the output of a list l for which no particular constraints are given (*cf.* the any_c on line 7). However, the continuation contract depends on the length of l. More precisely, the continuation describes an endpoint that must be used for sending

[1] The type inferred by OCaml is in fact an encoded form of the type shown here. FuSe comes along with a utility that pretty prints encoded OCaml types into the more readable syntax that we have used.

another endpoint with contract `fwd (List.length l)` before being closed. The expression `fwd n` (lines 2–6) yields a contract for an endpoint that must be used for sending exactly `n` messages. This is achieved using the `left_choice_c` and `right_choice_c` contracts. If `n > 0` the forwarder must select the left branch and then send one message followed by `n - 1` more (line 4). If `n` is 0 the forwarder must select the right branch and then terminate the interaction (line 6).

We conclude with two remarks. First, the contract `forwarder_c` allows `forwarder_body` to change the forwarded messages as long as their number is the same as the length of `l`. Parametricity only guarantees that the forwarded elements come from `l`. A stronger version of the contract could be written by making `fwd` a function that operates over the list of remaining messages and adding a condition over sent messages that checks whether the forwarded message is on the list of messages to be delivered. Second, the `forwarder_c` contract ties the behavior of a process on one session (`y`) to data (`l`) exchanged over a different session (`x`). Therefore, besides providing a means for specifying precise constraints over messages exchanged in one session, contracts for higher-order sessions enable the specification of inter-session dependencies.

7 RELATED WORK

Blame Correctness. Blame assignment with higher-order and dependent contracts is subtle because contract verification is deferred until the monitored object is actually used. In fact, alternative monitoring strategies called *lax*, *picky* and *indy* have been proposed [Blume and McAllester 2006; Dimoulas et al. 2011; Greenberg et al. 2012]. Such design choices pose the question of whether a particular strategy is reasonable, *i.e.*, if it assigns blames correctly. This problem has been addressed either by showing that principals that can be blamed have responsibility in a contract violation [Dimoulas et al. 2011, 2012] or by providing a characterization of components that *satisfy* a given contract [Blume and McAllester 2006; Dimoulas and Felleisen 2011; Findler and Blume 2006]. Given a semantics for contracts, contract satisfaction addresses the problem of showing that a contract system is sound and complete with respect to that semantics, *i.e.* that each blame assigned by the system corresponds to a contract violation and each contract violation is detected by the system. Our blame soundness result partially addresses this problem by showing that the contract system does not report blames when a component does not violate its contracts. The exact relationship between blame soundness and contract satisfaction remains to be fully understood.

Contracts have been integrated into type systems as refinement types, which are checked dynamically via cast insertion, like the hybrid types of Knowles and Flanagan [2010]. The relation between contracts and hybrid types has been studied by Greenberg et al. [2012] and Gronski and Flanagan [2007]. Wadler and Findler [2009] provide a unifying view with the blame calculus, but leave dependent contracts for future work. The blame calculus allows for a simple characterization of correct component when typed and untyped code are mixed. The blame theorem shows that blames are always assigned to the lesser-typed components.

Although contracts in λCoS do not interact with the type system, our notions of local correctness and blame soundness have been inspired by Wadler and Findler [2009] and Wadler [2015]. Communication over a monitored endpoint is the only point in the execution of a program that can originate a blame. We interpret communication as an implicit cast in which the message is coerced to the type specified in the contract wrapping the endpoint. However, contract satisfaction is verified dynamically in our approach. We leave the problem of static verification of session contracts, e.g., along the lines of Nguyen et al. [2014], as future work.

Behavioral/Temporal Higher-order Contracts. The interplay between higher-order contracts and behavioral/temporal aspects of modules, such as restricting the order in which functions can be invoked, has been previously addressed by Disney et al. [2011] and Scholliers et al. [2015]. In

both approaches, the enforcement of temporal constraints is done dynamically and the contract language allows for the specification of both allowed and disallowed traces. On the contrary, in λCoS, the usage of endpoints is regulated by a combination of static and dynamic constraints: static constraints are enforced by session types, which guarantee that processes use session endpoints according to their protocol; dynamic constraints, which concern the content of exchanged messages and may affect the selection and availability of choices and branches (Section 6.3), are checked at runtime by monitors. None of the previous works on behavioral/temporal contracts [Disney et al. 2011; Scholliers et al. 2015] provides a characterisation of module correctness or a formal statement about the correctness of blame assignment akin to our blame soundness result (Section 5).

Swords et al. [2015] proposed λ_{CC}, a language tailored to the implementation of alternative approaches to monitoring, and discussed a possible implementation of the temporal contracts of Disney et al. [2011] on top of λ_{CC}. An interesting question for future work is whether the primitives of λ_{CC} allows for a convenient implementation of λCoS to avoid monitor migration in delegations.

Contracts and Affine/Linear Types. Contract monitoring may duplicate values and discard contracts (Tables 2 and 3). These aspects have not been investigated elsewhere and affect the typing of contract constructors when the type system is substructural. If we allowed contracts to use linear resources, the monitoring of a non-linear object with a contract using linear resources would affect the behavior of the object, which would become linear itself. In this sense, our contracts conform to the definition of "chaperone contracts" given by Strickland et al. [2012], where chaperones are not allowed to affect the behavior of monitored objects.

Tov and Pucella [2010] use stateful contracts for controlling the usage of affine functions (functions that can be applied at most once) when these flow into a region of the program that uses a conventional (*i.e.*, non substructural) type system.

Projections and Duality. Contract duality is tightly related to session type duality [Honda 1993; Honda et al. 1998] and should not be confused with the client and server projections of Findler and Blume [2006]. The point is that dual contracts describe the same obligations in a session with respect to complementary directions of the messages being exchanged, whereas projection separates the obligations for values flowing into and out of a monitored expression.

Contracts for Sessions. Bocchi et al. [2010] and Toninho and Yoshida [2016] extend global types with assertions to specify constraints on values communicated in a multiparty session. These approaches are based on a top-down methodology whereby the whole multiparty interaction is designed at once and then projected on the single participants of the session. We work with binary sessions only. Our contracts can be applied gradually to arbitrary subsets of interacting modules and can be used to describe whole protocols or only fragments thereof. The session type is inferred automatically from the structure of the program. Bocchi et al. [2010] use a decidable assertion logic and Toninho and Yoshida [2016] use dependent session types along the lines of Toninho et al. [2011]. These choices make it possible to verify the correctness of participants statically. The soundness result rests on the assumption that all processes participating in a session are well typed, requiring no runtime monitoring. This assumption is relaxed in some works [Bocchi et al. 2013; Chen et al. 2011], where participants may misbehave and a monitor is used to suppress messages that violate the protocol. Neither higher-order sessions nor blame assignment are considered in these works.

Monitoring of untrusted processes has been also used [Bartoletti et al. 2013, 2012; Jia et al. 2016] to make sure that processes follow the intended protocol (session fidelity) and to assign blame if this is not the case. In these works monitoring does not concern the content of messages and

blame freedom can be guaranteed by typing. In λCoS session fidelity is ensured by typing whereas contracts specify conditions on the content of exchanged messages.

Thiemann [2014] studies a gradual type system for sessions with explicit coercions for branches and choices. Again, the dynamic checks concern session fidelity and particularly the branching structure of protocols.

8 CONCLUDING REMARKS

Building on contracts for higher-order functions pioneered by Findler and Felleisen [2002] and later extended to mutable objects by Strickland et al. [2012] we have defined and implemented a monitoring technique for higher-order sessions. Its characterizing aspect is that the contract wrapping a session endpoint must be dynamically updated at runtime, as the session progresses.

The setting provided us with the opportunity to investigate the ramifications of contract monitoring in the presence of linear resources. In particular, we have seen that session endpoints should not occur in contracts, which is in line with the observation that contracts should be "pure" [Meyer 1992]. Indeed, the operations involved with session endpoints are inherently impure.

We have proved a blame soundness result stating that modules which do not violate contracts are not blamed. The result rests on the key assumption that messages are either session endpoints or unlimited values (*i.e.* data and functions not containing session endpoints). This assumption allows us to tackle the higher-order case through delegation, which is the idiomatic form of mobility in sessions. Extending the blame soundness result to a setting where functions may contain session endpoints is an intriguing future development.

We intend to incorporate our proof-of-concept implementation of chaperone contracts for sessions into FuSe [Padovani 2017]. In this respect, the GADT presented in Section 6 suffers from two important limitations. First, the GADT does not allow the specification of contracts for structured data types (such as pairs) containing session endpoints. This can be overcome either by adding dedicated constructors corresponding to these data types [Hinze et al. 2006] or by adding a single, general-purpurse constructor that carries a user-provided, type-specific wrapper function. The second current limitation is that recursively defined constracts (such as `forwarder_c` in Section 6.3) must be finite in general. The finiteness of `forwarder_c` was guaranteed by the fact that its structure is isomorphic to that of a (finite) list. However, infinite protocols are commonly found in practice. In these cases, a contract definition relying on OCaml's support for recursion would diverge. This limitation can be lifted by adding a constructor to the GADT that lazily computes the fixpoint of a recursively defined contract. Once all these ingredients are in place, a proper evaluation of the overhead induced by our technique for contract monitoring is in order.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their detailed and thoughtful comments. The first author has been partially supported by World Wide Style project of the University of Torino (second edition) and Fondazione CRT.

REFERENCES

Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. 2013. Honesty by Typing. In *Proceedings of FMOODS/FORTE'13 (LNCS 7892)*. Springer, 305–320.

Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. 2012. Contract-Oriented Computing in CO2. *Scientific Annals of Computer Science* 22, 1 (2012), 5–60.

Matthias Blume and David A. McAllester. 2006. Sound and complete models of contracts. *Journal of Functional Programming* 16, 4-5 (2006), 375–414.

Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. 2013. Monitoring Networks through Multiparty Session Types. In *Proceedings FMOODS/FORTE'13 (LNCS 7892)*. Springer, 50–65.

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *Proceedings of CONCUR'10 (LNCS 6269)*. Springer, 162–176.

Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. 2011. Asynchronous Distributed Monitoring for Multiparty Session Enforcement. In *Proceedings of TGC'11 (LNCS 7173)*. Springer, 25–45.

Christos Dimoulas and Matthias Felleisen. 2011. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems* 33, 5 (2011), 16.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct blame for contracts: no more scapegoating. In *Proceedings of POPL'11*. ACM, 215–226.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Proceedings of ESOP'12 (LNCS 7211)*. Springer, 214–233.

Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal higher-order contracts. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 176–188.

Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *Proceedings of FLOPS'06 (LNCS 3945)*. Springer, 226–241.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of ICFP'02*. ACM, 48–59.

Jacques Garrigue and Jacques Le Normand. 2011. Adding GADTs to OCaml: the direct approach. In *Proceedings of ACM SIGPLAN Workshop on ML*.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-calculus. *Acta Informatica* 42, 2-3 (2005), 191–225.

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular session types for distributed object-oriented programming. In *Proceedings of POPL'10*. ACM, 299–312.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2012. Contracts made manifest. *Journal of Functional Programming* 22, 3 (2012), 225–274.

Jessica Gronski and Cormac Flanagan. 2007. Unifying Hybrid Types and Contracts. In *Proceedings of TFP'07*, Vol. 8. Intellect, 54–70.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed Contracts for Functional Programming. In *Proceedings of FLOPS'06 (LNCS 3945)*. Springer, 208–225.

Kohei Honda. 1993. Types for dyadic interaction. In *Proceedings of CONCUR'93 (LNCS 715)*. Springer, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of ESOP'98 (LNCS 1381)*. Springer, 122–138.

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and blame assignment for higher-order session types. In *Proceedings of POPL'16*. ACM, 582–594.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Transactions on Programming Languages and Systems* 32, 2 (2010).

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.

Hernán Melgratti and Luca Padovani. 2017. Chaperone Contracts for Higher-Order Sessions. (2017). Available at https://hal.archives-ouvertes.fr/hal-01545695 (last accessed Jun 2017).

Bertrand Meyer. 1992. Design by Contract. In *Advances in Object-oriented Software Engineering*. Prentice-Hall, 1–50.

Phúc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft contract verification. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 139–152.

Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming* 27 (2017).

António Ravara and Vasco Thudichum Vasconcelos. 2000. Typing Non-uniform Concurrent Objects. In *Proceedings of CONCUR'00 (LNCS 1877)*. Springer, 474–488.

Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2015. Computational contracts. *Science of Computer Programming* 98 (2015), 360–375.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *Proceedings of OOPSLA'12*. ACM, 943–962.

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2015. Expressing contract monitors as patterns of communication. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 387–399.

Peter Thiemann. 2014. Session Types with Gradual Typing. In *Proceedings of TGC'14 (LNCS 8902)*. Springer, 144–158.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *Proceedings of PPDP'11*. ACM, 161–172.

Bernardo Toninho and Nobuko Yoshida. 2016. Certifying Data in Multiparty Session Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (LNCS 9600)*. Springer, 433–458.

Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Proceedings of ESOP'10 (LNCS 6012)*. Springer, 550–569.

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of POPL'11*. ACM, 447–458.

Philip Wadler. 2015. A Complement to Blame. In *Proceedings of SNAPL'15 (LIPIcs 32)*. Schloss Dagstuhl, 309–320.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of ESOP'09 (LNCS 5502)*. Springer, 1–16.

Received February 2017; revised May 2017; accepted June 2017