



Recursive Logical Relations for Intuitionistic Linear Logic Session Types^{*}

Stephanie Balzer¹, Farzaneh Derakhshan², Robert Harper¹, and Yue Yao¹

¹ Carnegie Mellon University, {balzers,rwh,yueyao}@cs.cmu.edu

² Illinois Institute of Technology, fderakhshan@illinoistech.edu

Abstract. Program *equivalence* is the heart of reasoning about and proving properties of programs. To assert noninterference, for example, a program is shown to be equivalent to itself up to the confidentiality level of an observer. A powerful enabler for such proofs are *logical relations*, which, guided by the type structure, prescribe when two programs are indistinguishable. Logical relations enjoy ample exploration in functional languages, including languages with general recursion and a higher-order store—yet logical relations for session types only exist for terminating languages. This paper scales logical relations to *general recursive* session types. It develops a logical relation for *progress-sensitive equivalence* for *intuitionistic linear logic session types*, tackling the challenges non-termination and concurrency pose. In particular, the relation only equates a diverging program with another diverging one and accounts for nondeterminism of scheduling. The logical relation has two distinguishing characteristics: it is (i) indexed with an *intuitionistic linear sequent*, validating *cut reductions* and affording *biorthogonal closure*, and (ii) bound by an *observation index*, stratifying the logical relation in the presence of recursion. Biorthogonal closure validates the logical relation, proving that the induced equivalence is *sound and complete* with regard to closure of weak bisimilarity under parallel composition. Soundness guarantees that the equivalence has enough discriminatory power, completeness ensures that it is maximally permissive. The logical relation is then put to test on the example of noninterference.

Keywords: Logical relations · Biorthogonality · Step-indexing · Intuitionistic linear logic session types · Progress-sensitive noninterference.

1 Introduction

Whether two programs are *equivalent* is at the heart of many program verification problems, such as proofs of parametricity, compiler correctness, and noninterference. This paper develops a *logical relation* to reason about program equivalence of *session-typed processes* and proves soundness and completeness of the relation via a *biorthogonality* argument.

^{*} The first two authors have equal contributions.

Message-passing concurrency. Rooted in process calculi [28, 44, 45], message-passing concurrency enjoys widespread adoption, including languages such as Erlang, Go, and Rust. A program in this setting amounts to a number of *processes* connected by *channels*, which compute by exchanging *messages* along these channels. Messages can even amount to channels themselves, giving rise to so-called *higher-order* channels, as present in the π -calculus [46, 57]. This feature is equally empowering as daunting because it changes the process topology dynamically. Originally untyped, the π -calculus [46] has gradually been enriched with types [57] to prescribe the kinds of messages that can be exchanged over a channel. More advanced type systems [33, 34, 36, 37] additionally assert correctness properties, such as deadlock freedom and data race freedom.

To prescribe not only the types of exchanged messages but also the *protocol* underlying the exchange, *session types* [29, 30] were introduced. Session types rely on a *linear* treatment of channels to model the state transitions induced by a protocol. This foundation manifests in a correspondence between linear logic and the session-typed π -calculus [12, 66], resulting in two families of session type languages: *intuitionistic linear logic session types (ILLST)* [12, 13, 63, 64] and *classical linear logic session types (CLLST)* [38, 40, 41, 66]. Due to their logical foundation well-typed ILLST/CLLST processes not only are protocol-compliant (a.k.a., preservation), but also free of data races and deadlocks (a.k.a., progress).

This paper studies logical relations for ILLST-typed processes. ILLST reject linear negation and distinguish the *provider* from the *client* side of a channel. Run-time configurations of processes thus give raise to a *rooted tree*, with a *child* node as the provider and the *parent* node as the client.

Program equivalence. Today's predominant techniques for reasoning about program equivalence of stateful programs are Kripke logical relations (KLRs) [3, 21, 31, 48, 52, 61] and bisimulations [39, 56, 59, 60]. KLRs tend to be used for sequential, ML-like languages, bisimulations for concurrent process calculi. KLRs are phrased by structural induction on types, bisimulations by coinduction. Hur et al. [32] note that KLRs and bisimulations have mutually disjoint strengths and weaknesses. Whereas KLRs naturally support higher-order features, but struggle with recursive types, bisimulations naturally support recursion, but struggle with higher-order features. To support recursive types KLRs employ step-indexing [1, 2, 4], complicating proofs with step arithmetic [10] and challenging transitivity of the logical relation [32].

This paper scales logical relations to session-typed concurrency in the presence of *general recursive types*, contributing a *recursive session logical relation (RSLR)* for intuitionistic linear logic session types (ILLST). In contrast to KLRs, which index the logical relation with the type of the considered expression, RSLRs index the logical relation with an *intuitionistic linear sequent*, denoting the types of the free channels along which the considered process configuration exchanges messages with the outside world. The usual term interpretation clause

$$(e_1; e_2) \in \mathcal{E}[\tau] \text{ iff } \dots$$

for expressions e_1 and e_2 and a type τ in KLRs thus takes the form

$$(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash A] \text{ iff } \dots$$

in RSLRs, for configurations $\mathcal{D}_1$ and $\mathcal{D}_2$ with the types of free channels denoted by the sequent $\Delta \Vdash A$. This generalization is necessitated by the underlying computational model (message-passing concurrency) and the goal to accommodate various program verification problems, including noninterference. The use of a sequent as an index makes explicit the *duality* of session types, allowing a type to be associated with two interpretations: as a *provider* of a session of that type and a *client*. For example, as a provider of a session $\&\{left:A, right:B\}$, two related configurations may *assume* to receive related messages, i.e., either they both receive *left* or *right*. As a client of such a session, conversely, related configurations must *assert* to either both send *left* or *right*. The latter condition, crucial for establishing noninterference, is lost when only indexing the logical relation with one type (i.e., the type of the provider). Duality is also central to enforcing a *resource* semantics for channels. For example, as a provider of a session $A \otimes B$, promising to send a channel of type A , related configurations must give up access to the sent channel, whereas a client of such a session will become its owner. Thanks to the validity of semantic cut (Lem. 2), the usual, single-type-index interpretation falls out for free as a special case.

To accommodate general recursive types, RSLRs use an *observation index* to stratify the logical relation. An observation index m is associated with the free channels in $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash A]^m$ and bounds the number of messages that $\mathcal{D}_1$ and $\mathcal{D}_2$ exchange along those channels. The index is thus associated with an *externally observable* event, symmetrically bounding both configurations. In contrast KLRs employ a step index [2,4,20], which is tied to the internal computation or unfolding steps of one of the two terms, making the relation asymmetric.

We establish several metatheoretic properties of RSLRs. In particular, we show that RSLRs entail an *equivalence relation*, whose proof of transitivity (Lem. 5) benefits from the extensionality of the observation index. Given the possibility of divergence, the equivalence is *progress-sensitive* [26] in that it equates a divergent program only with another diverging one. We also show that RSLRs are *sound and complete* (Thm. 1) with regard to closure of weak *bisimilarity* under parallel composition, using a *biorthogonality* argument à la Pitts [51]. The *duality* of session types establishes a natural connection to biorthogonality. To put the RSLR to test, we employ it to verify *progress-sensitive noninterference*.

Contributions. In summary, this paper makes the following contributions:

- *Recursive session logical relation (RSLR)*, a binary logical relation for *progress-sensitive equivalence* of ILLST processes with general recursion, featuring an *intuitionistic linear sequent* and *observation index*, to capture duality and to support general recursion without compromising extensionality, resp.
- Closure of the RSLR under parallel composition, a.k.a., *semantic cut* (Lem. 2).
- RSLR induces an *equivalence relation*, including *transitivity* (Lem. 5).
- Soundness and completeness of the RSLR with regard to *weak asynchronous bisimilarity*, a.k.a., *adequacy* (Lem. 6).

- Biorthogonal ($\top\top$ -) closure of the induced equivalence relation (Thm. 1), guaranteeing that the equivalence has enough discriminatory power (soundness), while being maximally permissive (completeness).
- Small case study applying the RSLR to *progress-sensitive noninterference*.

Technical report. The complete formalization and all proofs are available in [5].

2 Background

This section familiarizes with intuitionistic linear logic session types (ILLST).

ILLST type system. Our development is based on a variant of ILLST languages [12, 13, 63, 64] that supports general recursive types [6, 64]. We refer to this language as **SESSION**.

SESSION's connectives are drawn from intuitionistic linear logic and obey the following grammar

$$A, B ::= \oplus\{\ell:A_\ell\}_{\ell \in L} \mid \&\{\ell:A_\ell\}_{\ell \in L} \mid A \otimes B \mid A \multimap B \mid 1 \mid Y,$$

where L ranges over finite, non-empty sets of labels denoted by ℓ and k , the primitive values in **SESSION**. Type variable Y is a fixed point whose definition $Y = A$ is collected in a global signature Σ . General recursive types are supported through this definition mechanism. They must be *contractive* [23], demanding a message exchange before recurring, and *equi-recursive* [16], avoiding explicit (un)fold messages and relating types up to their unfolding.

Process terms are typed using the *intuitionistic sequent*

$$\Omega \vdash_\Sigma P :: x:A$$

to be read as “process P provides a session of type A along channel variable x , given the typing of sessions offered along channel variables in Ω and given the type and process definitions in Σ ”. Ω is a linear context, consisting of assumptions $y_i:B_i$, indicating for each channel variable y_i its session type B_i . The signature Σ contains global type and process definitions and facilitates recursive type and process definitions. Bound channel variables are substituted with channels that are created at run-time upon process spawning. When the distinction is clear from the context we refer to either as “channels” for brevity.

ILLST rejects linear negation, allowing the singleton right-hand side of the sequent to be interpreted as the type of the *providing* process, which, conversely, is the *client* of the sessions in Ω . Channels can thus be typed with the type of the providing process, rather than typing the two channel endpoints dually, as done in classical linear logic session types (CLLST) [40, 41, 66]. To express the *duality* in behavior the ILLST type system is given as *sequent calculus*, comprising both a *right* and a *left* rule per connective. Right rules define a communication from the point of view of the provider, left rules from the point of view of the client.

Fig. 1 summarizes the typing rules. Cut reduction in the sequent calculus invites a computational, bottom-up reading of the rules, where the type of the conclusion denotes the protocol state of the provider before the message exchange and the type of the premise the protocol state after the message exchange. The polarity of type moreover determines the direction of communication: for positive connectives, the provider sends and the client receives, for negative connectives, the provider receives and the client sends. We review each rule in turn next.

$$\begin{array}{c}
\frac{\Omega \vdash_{\Sigma} P :: x:A_k \quad k \in L}{\Omega \vdash_{\Sigma} x.k; P :: x: \oplus \{\ell:A_{\ell}\}_{\ell \in L}} \oplus R \\
\\
\frac{\Omega, x:A_k \vdash_{\Sigma} Q_k :: y:C \quad \forall k \in L}{\Omega, x: \oplus \{\ell: A_{\ell}\}_{\ell \in L} \vdash_{\Sigma} \mathbf{case} x(\ell \Rightarrow Q_{\ell})_{\ell \in L} :: y:C} \oplus L \\
\\
\frac{\Omega \vdash_{\Sigma} Q_k :: x:A_k \quad \forall k \in L}{\Omega \vdash_{\Sigma} \mathbf{case} x(\ell \Rightarrow Q_{\ell})_{\ell \in I} :: x:\&\{\ell: A_{\ell}\}_{\ell \in L}} \& R \quad \frac{\Omega, x:A_k \vdash_{\Sigma} P :: y:C \quad k \in L}{\Omega, x:\&\{\ell: A_{\ell}\}_{\ell \in I} \vdash_{\Sigma} x.k; P :: y:C} \& L \\
\\
\frac{\Omega \vdash_{\Sigma} P :: x:B}{\Omega, z:A \vdash_{\Sigma} \mathbf{send} z x; P :: x:A \otimes B} \otimes R \quad \frac{\Omega, z:A, x:B \vdash_{\Sigma} P :: y:C}{\Omega, x:A \otimes B \vdash_{\Sigma} z \leftarrow \mathbf{recv} x; P :: y:C} \otimes L \\
\\
\frac{\Omega, z:A \vdash_{\Sigma} P :: x:B}{\Omega \vdash_{\Sigma} z \leftarrow \mathbf{recv} x; P :: x:A \multimap B} \multimap R \quad \frac{\Omega, x:B \vdash_{\Sigma} P :: y:C}{\Omega, z:A, x:A \multimap B \vdash_{\Sigma} \mathbf{send} z x; P :: y:C} \multimap L \\
\\
\frac{}{\cdot \vdash_{\Sigma} \mathbf{close} x :: x:1} 1R \quad \frac{\Omega \vdash_{\Sigma} Q :: y:C}{\Omega, x:1 \vdash_{\Sigma} \mathbf{wait} x; Q :: y:C} 1L \\
\\
\frac{\Omega'_1 \vdash X = P :: x':A \in \Sigma \quad \Omega_1, x:A \Vdash \gamma :: \Omega'_1, x':A \quad \Omega_2, x:A \vdash_{\Sigma} Q :: y:C}{\Omega_1, \Omega_2 \vdash_{\Sigma} (x \leftarrow X[\gamma] \leftarrow \Omega_1); Q :: y:C} \text{SPAWN} \\
\\
\frac{z':C \vdash F_Y = \mathbf{Fwd}_{C,y' \leftarrow z'} :: y':C \in \Sigma \quad Y = C \in \Sigma \quad z:C, y:C \Vdash \gamma :: z':C, y':C}{z:Y \vdash_{\Sigma} F_Y[\gamma] :: y:C} \text{D:FWD} \\
\\
\frac{Y = A \in \Sigma \quad \Omega \vdash_{\Sigma} P :: x:A}{\Omega \vdash_{\Sigma} P :: x:Y} \text{TVAR}_R \quad \frac{Y = A \in \Sigma \quad \Omega, x:A \vdash_{\Sigma} P :: z:C}{\Omega, x:Y \vdash_{\Sigma} P :: z:C} \text{TVAR}_L \\
\\
\frac{}{\Vdash_{\Sigma} (\cdot) \mathbf{sig}} \Sigma_1 \quad \frac{\Vdash_{\Sigma} A \mathbf{contr} \quad \Vdash_{\Sigma} \Sigma' \mathbf{sig}}{\Vdash_{\Sigma} Y = A, \Sigma' \mathbf{sig}} \Sigma_2 \quad \frac{\Omega \vdash_{\Sigma} P :: x:A \quad \Vdash_{\Sigma} \Sigma' \mathbf{sig}}{\Vdash_{\Sigma} \Omega \vdash_{\Sigma} X = P :: x:A, \Sigma' \mathbf{sig}} \Sigma_3
\end{array}$$

Fig. 1: Process term typing rules and signature checking rules of **SESSION**.

The additive connectives $\oplus\{\ell:A_{\ell}\}_{\ell \in L}$ and $\&\{\ell:A_{\ell}\}_{\ell \in L}$ denote *labelled choices* of sessions A_{ℓ} , where the labels ℓ range over the finite, non-empty set L . The two

connectives differ in who sends a label and thus makes a choice. For $\oplus\{\ell:A_\ell\}_{\ell \in L}$, the provider chooses, for $\&\{\ell:A_\ell\}_{\ell \in L}$, the client chooses, giving the connectives the names *internal choice* and *external choice*, resp. The duality of behavior of a provider and client is conveyed by the process terms of the right and left rules, resp. For example, a provider of an internal choice (rule $\oplus R$) executes the process term $x.k;P$ to send the label k along its providing channel x , after which it will continue with P of type A_k . Conversely, a client of an internal choice (rule $\oplus L$) executes the process term $\text{case } x(\ell \Rightarrow Q_\ell)_{\ell \in L}$ to receive any label $k \in L$ along x , after which it will continue with the chosen branch Q_k of type A_k .

The multiplicative connectives $A \otimes B$ (tensor) and $A \multimap B$ (lolli) allow channels themselves to be sent over channels, changing the process topology at run-time. These connectives endow SESSION with *higher-order channels*, making channels first-class values. Again, the two connectives differ in who sends a channel: for $A \otimes B$, the provider sends, for $A \multimap B$, the client sends. Due to linearity, absence of weakening specifically, the sender loses access to the sent channel in its continuation; as can be seen in the premises of rules $\otimes R$ and $\multimap L$. We use $\leftarrow$ to denote variable binding. For example, the process term $z \leftarrow \text{recv } x; P$ in the conclusion of rule $\otimes L$ binds the received channel to y , with scope P .

The unit of $\otimes$, 1 , allows a process to terminate. Here, the provider sends a close message, **close** x , along its providing channel x (rule $1R$), awaited by the client. Due to absence of weakening, rule $1R$ demands that the typing context Ω be empty, ensuring that no channels become orphans. Consequently, the client continuation Q loses access to x after receipt of the close message (rule $1L$).

Rule SPAWN types a process that spawns another process using the process definition X . The invocation $x \leftarrow X[\gamma] \leftarrow \Omega_1$ binds the newly spawned process to x and gives it the argument channels Ω_1 . After the invocation, the process continues with executing Q , now with $x:A$ in its context. For the invocation to succeed, a corresponding process definition $\Omega'_1 \vdash X = P :: x':A$ must exist in the signature Σ . The definition associates a name X with the process term P and indicates the names and types of argument channels Ω'_1 and the name and type of the providing channel $x':A$. To account for the difference in variable names, the programmer must provide a corresponding mapping $\Omega_1, x:A \Vdash \gamma :: \Omega'_1, x':A$ as part of the invocation, assigning to each variable in the definition $\Omega'_1, x':A$ a corresponding variable of the same type in the invocation $\Omega_1, x:A$.

Rule D:FWD allows a process to delegate requests to its singleton child z by passing any messages received along y to z and vice versa. The rule uses a forwarder definition $z':C \vdash F_Y = \text{Fwd}_{C,y' \leftarrow z'} :: y':C$, for the type definition $Y = C$, both to be present in the signature Σ . A corresponding forwarder $\text{Fwd}_{C,y' \leftarrow z'}$ of type C is defined by structural induction on the type of every user-provided type definition, amounting to an *identity expansion*, and can be automatically generated. In combination with rule D:FWD, we get a coinductive definition (see [5] for details). The programmer can use the syntactic sugar $y \leftarrow z$ instead, which can be expanded into the corresponding forwarder.

Rules TVAR_R and TVAR_L allow us to unfold type definitions. The rules insist that a corresponding definition exists in the global signature $Y = A \in \Sigma$.

Rules Σ_1 , Σ_2 , and Σ_3 type check the signature itself. Rule Σ_2 type checks type definitions, requiring the type be contractive ($\Vdash_{\Sigma} A \text{ contr}$) and thus not a type variable itself [23]. Rule Σ_3 type checks the body P of a process definition X , whose existence is assumed when spawning the process in rule **SPAWN**.

We illustrate the typing rules on a simple PIN-based authentication example, shown in Fig. 2. Fig. 2 introduces two recursive type definitions, **pin** and **ver**, as well as two process definitions, **Verifier** and **Pin**. A verifier expects to receive a PIN channel from its client, which it validates. If validation is successful, the verifier sends the message *succ*, otherwise the message *fail* to the client. In either case, the verifier also returns the PIN channel. The PIN itself is encoded as an internal choice of labels tok_i , with one being the correct security token.

$$\begin{aligned}
 \text{ver} &= \text{pin} \multimap \oplus\{succ:\text{pin} \otimes \text{ver}, fail:\text{pin} \otimes \text{ver}\} & \text{pin} &= \oplus\{tok_1:\text{pin}, \dots, tok_n:\text{pin}\} \\
 \cdot \vdash \text{Verifier} :: x:\text{ver} &= (u \leftarrow \text{recv } x \quad // \quad u:\text{pin} \vdash x: \oplus\{succ : \text{pin} \otimes \text{ver}, fail:\text{pin} \otimes \text{ver}\} \\
 &\quad \text{case } u(tok_j \Rightarrow x.succ; \text{send } u x; x \leftarrow \text{Verifier} \leftarrow \cdot \\
 &\quad | tok_{i \neq j} \Rightarrow x.fail; \text{send } u x; x \leftarrow \text{Verifier} \leftarrow \cdot)) \\
 \cdot \vdash \text{Pin} :: u:\text{pin} &= (u.tok_j; u \leftarrow \text{Pin} \leftarrow \cdot)
 \end{aligned}$$

Fig. 2: Example: PIN-based authentication.

ILLST configuration typing At runtime, **SESSION** programs become configurations of processes, forming *rooted trees* and obeying the following grammar:

$$\begin{aligned}
 \mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{T} &::= \mathbf{proc}(x_{\alpha}, \hat{\delta}(P)) \mathcal{C} \mid \mathbf{msg}(M) \mathcal{C} \mid \cdot \\
 M &::= x_{\alpha}.k \mid \mathbf{send } z_{\beta} x_{\alpha} \mid \mathbf{close } y_{\alpha}
 \end{aligned}$$

Configurations consist of a set of processes of the form $\mathbf{proc}(y_{\alpha}, \hat{\delta}(P))$ and a set of messages of the form $\mathbf{msg}(M)$. Every spawn results in a process $\mathbf{proc}(y_{\alpha}, \hat{\delta}(P))$, every send in a message $\mathbf{msg}(M)$, making **SESSION**'s dynamics asynchronous. The metavariable y_{α} in $\mathbf{proc}(y_{\alpha}, \hat{\delta}(P))$ denotes the process' providing channel and the metavariable P the process' source code. Run-time channels y_{α} can be distinguished from channel variables y by their generation subscript α , whose meaning we clarify in § 2. The substitution δ maps variables to channels; $\hat{\delta}(P)$ yields the term with all free channel variables substituted by channels.

The configuration typing rules of **SESSION** are given in Fig. 3, using the judgment $\Delta_0 \Vdash \mathcal{C} :: \Delta$. We allow configurations to have *free channels*, which amount to Δ_0 and Δ for the given judgment. The channels in Δ comprise the channels for which there exist providing processes in the configuration $\mathcal{C}$, the channels in Δ_0 comprise the channels for which there exist client processes in the configuration $\mathcal{C}$. It may be surprising that we refer to these channels as free, given that they occur in $\mathcal{C}$. Would linearity not imply that they occur “exactly once”, and thus must be bound? It is more fruitful to think of channels as shared

resources, like memory locations, *shared* among processes. ILLST then ensures that at most two processes, a provider and a client, share a channel at any point in time, where each "owns" one endpoint of (i.e., a reference to) the channel. The careful accounting of "resources" is visible in the configuration typing rules. For example, rule **proc** insists that for all the channel endpoints used by process $\mathbf{proc}(x_\alpha, \hat{\delta}(P))$ there exists either a provider in the sub-configuration $\mathcal{C}$, making the channel bound, or the endpoint is free and occurs in Δ'_0 .

$$\begin{array}{c}
\frac{}{x_\alpha:A \Vdash \cdot :: x_\alpha:A} \mathbf{emp}_1 \quad \frac{}{\cdot \Vdash \cdot :: \cdot} \mathbf{emp}_2 \quad \frac{\Delta_0 \Vdash \mathcal{C} :: \Delta \quad \Delta'_0 \Vdash \mathcal{C}_1 :: x_\alpha:A}{\Delta_0, \Delta'_0 \Vdash \mathcal{C} \mathcal{C}_1 :: \Delta, x_\alpha:A} \mathbf{comp} \\
\\
\frac{\Delta_0 \Vdash \mathcal{C} :: \Delta \quad \Delta'_0, \Delta, x_\alpha:A \vdash \delta :: \Omega'_0, \Omega, x:A \quad \Omega'_0, \Omega \vdash_\Sigma P :: x:A}{\Delta_0, \Delta'_0 \Vdash \mathcal{C} \mathbf{proc}(x_\alpha, \hat{\delta}(P)) :: x_\alpha:A} \mathbf{proc} \\
\\
\frac{\Delta_0 \Vdash \mathcal{C} :: \Delta \quad \Delta'_0, \Delta \vdash M :: x_\alpha:A}{\Delta_0, \Delta'_0 \Vdash \mathcal{C} \mathbf{msg}(M) :: x_\alpha:A} \mathbf{msg} \\
\\
\frac{k \in L}{y_{\alpha+1}:A_k \vdash y_\alpha.k :: y_\alpha:\oplus \{\ell:A_\ell\}_{\ell \in L}} \mathbf{m}_{\oplus R} \quad \frac{k \in L}{x_\alpha:\&\{\ell:A_\ell\}_{\ell \in I} \vdash x_\alpha.k :: x_{\alpha+1}:A_k} \mathbf{m}_{\&L} \\
\\
\frac{}{z_\beta:A, y_{\alpha+1}:B \vdash \mathbf{send} z_\beta y_\alpha :: y_\alpha:A \otimes B} \mathbf{m}_{\otimes R} \\
\\
\frac{}{z_\beta:A, x_\alpha:A \multimap B \vdash \mathbf{send} z_\beta x_\alpha :: x_{\alpha+1}:B} \mathbf{m}_{\multimap L} \quad \frac{}{\cdot \vdash \mathbf{close} y_\alpha :: y_\alpha:1} \mathbf{m}_{1R}
\end{array}$$

Fig. 3: Configuration typing rules of SESSION.

ILLST asynchronous dynamics SESSION's asynchronous dynamics is given in Fig. 4. It is phrased as multiset rewriting rules [14]. Multiset rewriting rules express the dynamics as state transitions between configurations $C \mapsto C'$ and are *local* in that they only mention the parts of a configuration they rewrite.

Being *asynchronous*, the dynamics implements sends by spawning off a message $\mathbf{msg}(M)$ that carries the sent message M . In rule $\otimes_{\text{snd}}$, for example, the provider $\mathbf{proc}(y_\alpha, \mathbf{send} x_\beta y_\alpha; P)$ generates the message $\mathbf{msg}(\mathbf{send} x_\beta y_\alpha)$, indicating that the channel x_β is sent over channel y_α . In rule $\otimes_{\text{rcv}}$, the client then consumes the message when receiving the channel. To preserve the invariant stipulated by typing that at most two processes share a channel at any point in time, a new providing channel must be generated for the continuation of the provider. This new channel must be linked to the old providing channel, otherwise proper sequencing of messages would be violated.

An elegant way to achieve proper sequencing are *channel generations*. Rather than creating an entirely fresh channel name for each send, we keep the name of a channel constant but increment its generation, provided as a subscript to

the channel name. For example, the provider $\mathbf{proc}(y_\alpha, \mathbf{send } x_\beta y_\alpha; P)$ in rule $\otimes_{\text{snd}}$ steps to its continuation $\mathbf{proc}(y_{\alpha+1}, [y_{\alpha+1}/y_\alpha]P)$, creating a new generation $\alpha + 1$ of its providing channel y_α . The receiving client process $\mathbf{proc}(u_\eta, w \leftarrow \mathbf{recv } y_\alpha; P)$ will then increment the channel generation of the carrier channel y_α upon receipt in its continuation $\mathbf{proc}(u_\eta, [x_\beta/w][y_{\alpha+1}/y_\alpha]P)$ (see rule $\otimes_{\text{rcv}}$). Like bound variables, bound channels are subject to α -variance and capture-avoiding substitution, as usual [57], with the understanding that only the name y of a channel y_α can be renamed, but not its generation.

$$\begin{array}{ll}
\text{SPAWN} & \mathbf{proc}(y_\alpha, (x \leftarrow X[\gamma] \leftarrow \Delta_1); Q) \mapsto \mathbf{proc}(x_0, \hat{\gamma}([x_0/x]P)) \mathbf{proc}(y_\alpha, [x_0/x]Q) \quad (\Omega'_1 \vdash X = P :: x' : B \in \Sigma) \\
& \quad \quad \quad (\Omega'_1 \Vdash \gamma : \Delta_1, \ x_0 \text{ fresh}) \\
\text{D:FWD} & \mathbf{proc}(y_\alpha, F_Y[\gamma]) \mapsto \mathbf{proc}(y_\alpha, \mathbf{Fwd}_{C, y' \leftarrow x'} :: y' : C \in \Sigma) \quad (x' : C \vdash F_Y = \mathbf{Fwd}_{C, y' \leftarrow x'} :: y' : C \in \Sigma) \\
& \quad \quad \quad (x', y' \Vdash \gamma : x_\beta, y_\alpha) \\
1_{\text{snd}} & \mathbf{proc}(y_\alpha, \mathbf{close } y_\alpha) \mapsto \mathbf{msg}(\mathbf{close } y_\alpha) \\
1_{\text{rcv}} & \mathbf{msg}(\mathbf{close } y_\alpha) \mathbf{proc}(x_\beta, \mathbf{wait } y_\alpha; Q) \mapsto \mathbf{proc}(x_\beta, Q) \\
\oplus_{\text{snd}} & \mathbf{proc}(y_\alpha, y_\alpha.k; P) \mapsto \mathbf{proc}(y_{\alpha+1}, [y_{\alpha+1}/y_\alpha]P) \mathbf{msg}(y_\alpha.k) \\
\oplus_{\text{rcv}} & \mathbf{msg}(y_\alpha.k) \mathbf{proc}(u_\eta, \mathbf{case } y_\alpha(\ell \Rightarrow P_\ell)_{\ell \in L} \mapsto \mathbf{proc}(u_\eta, [y_{\alpha+1}/y_\alpha]P_k) \\
\&_{\text{snd}} & \mathbf{proc}(y_\alpha, x_\beta.k; P) \mapsto \mathbf{msg}(x_\beta.k) \mathbf{proc}(y_\alpha, [x_{\beta+1}/x_\beta]P) \\
\&_{\text{rcv}} & \mathbf{proc}(y_\alpha, \mathbf{case } y_\alpha(\ell \Rightarrow P_\ell)_{\ell \in L} \mapsto \mathbf{msg}(y_\alpha.k) \mapsto \mathbf{proc}(y_{\alpha+1}, [y_{\alpha+1}/y_\alpha]P_k) \\
\otimes_{\text{snd}} & \mathbf{proc}(y_\alpha, \mathbf{send } x_\beta y_\alpha; P) \mapsto \mathbf{proc}(y_{\alpha+1}, [y_{\alpha+1}/y_\alpha]P) \mathbf{msg}(\mathbf{send } x_\beta y_\alpha) \\
\otimes_{\text{rcv}} & \mathbf{msg}(\mathbf{send } x_\beta y_\alpha) \mathbf{proc}(u_\eta, w \leftarrow \mathbf{recv } y_\alpha; P) \mapsto \mathbf{proc}(u_\eta, [x_\beta/w][y_{\alpha+1}/y_\alpha]P) \\
\multimap_{\text{snd}} & \mathbf{proc}(y_\alpha, \mathbf{send } x_\beta u_\eta; P) \mapsto \mathbf{msg}(\mathbf{send } x_\beta u_\eta) \mathbf{proc}(y_\alpha, [u_{\gamma+1}/u_\gamma]P) \\
\multimap_{\text{rcv}} & \mathbf{proc}(y_\alpha, w \leftarrow \mathbf{recv } y_\alpha; P) \mathbf{msg}(\mathbf{send } x_\beta y_\alpha) \mapsto \mathbf{proc}(y_{\alpha+1}, [x_\beta/w][y_{\alpha+1}/y_\alpha]P)
\end{array}$$

Fig. 4: Asynchronous dynamics of SESSION.

3 Recursive session logical relation (RSLR)

This section develops a logical relation for ILLST with general recursive types (§ 3.1) and shows that the logical relation is closed under parallel composition (§ 3.2), amounts to an equivalence relation (§ 3.3), and adequate, i.e., a weak asynchronous bisimilarity (§ 3.4). The logical relation is shown to have enough discriminatory power (soundness), while being maximally permissive (completeness) via a biorthogonality argument (Thm. 1).

3.1 Logical relation

Logical relations have been developed for session types [11, 17–19, 27, 49, 50, 54], both in unary and binary forms, but *without* considering general recursion. We contribute a logical relation for ILLST with support of *general recursive types*. In devising the resulting *recursive session logical relation (RSLR)*, we had to tackle the following challenges:

- *Well-foundedness*: Recursive types mandate use of a measure like step-indexing or later modalities [2,4,20] to keep the logical relation well-founded. Our logical relation makes use of a more *extensional* notion, an *observation index*, which bounds the number of messages exchanged with the outside world along the free channels of a configuration. The observation index facilitates proofs of various results shown in this section.
- *Non-termination*: In the presence of general recursive types, divergence is a possible outcome. Our logical relation is *progress-sensitive* [26] with regard to divergence; it equates a divergent program only with another diverging one.
- *Nondeterminism*: Although ILLST are confluent, processes run *concurrently*. As a result, the logical relation has to account for the relatedness of messages that may not be sent in the same order due to nondeterministic scheduling.

A logical relation for message-passing concurrency. It may be helpful to pause a moment and ask what a logical relation for session-typed processes should amount to. Logical relations for functional languages relate terms by defining their equality at their type when evaluated to values. This can be phrased in terms of two mutually recursive relations, a value interpretation $\mathcal{V}[\![\tau]\!]$ and a term interpretation $\mathcal{E}[\![\tau]\!]$, defined by structural induction on the type τ . The former defines equality of values by inducting over a type, with values at ground type forming the base cases, and the latter steps the terms until they reach a value, demanding that these must be related by the value interpretation. In imperative languages, logical relations are enriched with Kripke possible worlds to model dynamically evolving shapes of storage [3, 21, 31, 48, 52, 61].

To answer our question, it is helpful to remind ourselves that logical relations are a means to prove observational equivalence [47], which equates two programs, if the same observations can be made about them. In a pure functional setting, the observables are the values to which the terms evaluate. In an imperative setting, the observables can additionally comprise the values of locations in the store. The observables in a message-passing concurrent setting are the messages that a configuration of processes exchanges with the outside world.

A logical relation for session-typed processes thus relates pairs of process configurations $(\mathcal{D}_1; \mathcal{D}_2)$, such that $\Delta \Vdash \mathcal{D}_1 :: x_\alpha:A$ and $\Delta \Vdash \mathcal{D}_2 :: x_\alpha:A$, demanding that the same messages can be observed along the free channels Δ and $x_\alpha:A$. Fig. 5 shows the resulting logical relation for well-typed process configurations in **SESSION**, which is indexed with an *intuitionistic linear sequent* $\Delta \Vdash A$, comprising the free channels. We find it convenient to distinguish a value interpretation $\mathcal{V}$ from a term interpretation $\mathcal{E}$. As expected, the term interpretation steps the configurations until they reach a “value”. But what does it mean for a configuration to be a value in a message-passing concurrent setting? It means that a configuration is *ready to send* or *receive* along any of the free channels in $\Delta \cup \{x_\alpha:A\}$. For any such channel, the term interpretation invokes the value interpretation, which demands that the exchanged messages be related, and then invokes the term interpretation for the configurations resulting after the message

exchange. The mutually recursive relations are defined by clauses of the form

$$(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta \Vdash A] \quad \text{iff} \quad \dots (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta' \Vdash A'].$$

In a terminating setting, the relations are defined *multiset induction over the structure of types of the free channels* [17] such that $\Delta' \Vdash A' < \Delta \Vdash A$. To support general recursive types, we will augment the relations with an observation index, as discussed shortly, yielding an inductive definition.

The sequent $\Delta \Vdash A$ singles out the typing A of the providing free channel of the configurations $\mathcal{D}_1$ and $\mathcal{D}_2$. If the providing free channel is not observable, we write $_$. We use the metavariable K to stand for either $x_\alpha.A$ or simply $_$. The ability to mark some free channels as unobservable becomes important when we apply the logical relation to noninterference in § 4.

The transition from the value to the term interpretation amounts to an *observation* of a message exchange along a free channel in $\Delta \Vdash K$. The exchange will advance the protocol state of the concerned process, and as a result transition the sequent to $\Delta' \Vdash K'$, describing the post-state of the configurations $(\mathcal{D}'_1; \mathcal{D}'_2)$ after the message exchange. The logical relation comprises two value interpretation clauses for each connective, one for the communications occurring on the *right* along K , and one for those occurring on the *left* along Δ .

Assume and assert—a pas de deux. A key characteristic of logical relations is *extensionality*. In a functional context this means that relatedness has to be shown for values of types in positive positions, but values of types in negative positions can be assumed to be related. This idea translates equally to a message-passing concurrent setting, embodied by the logical relation shown in Fig. 5 as:

- *Positive types*: *assert* sending of related messages when communicating on the *right*; *assume* receipt of related messages when communicating on the *left*.
- *Negative types*: *assume* receipt of related messages when communicating on the *right*; *assert* sending of related messages when communicating on the *left*.

At base types (1 , $\oplus$, and $\&$), relatedness of messages means that the same messages are being exchanged. For example, clause (2) in Fig. 5 for $\oplus$ -right asserts existence of messages $\mathbf{msg}(y_\alpha.k_1)$ and $\mathbf{msg}(y_\alpha.k_2)$ in $\mathcal{D}_1$ and $\mathcal{D}_2$, resp., such that $k_1 = k_2$. Conversely, clause (3) in Fig. 5 for $\&$ -right assumes receipt of messages $\mathbf{msg}(y_\alpha.k_1)$ and $\mathbf{msg}(y_\alpha.k_2)$ to be added to $\mathcal{D}_1$ and $\mathcal{D}_2$ in the post-states, resp., such that $k_1 = k_2$, for arbitrary k_1 and k_2 .

For higher-order types ($\otimes$ and $\multimap$) relatedness means that future observations to be made along the exchanged channels must be related. For example, clause (10) in Fig. 5 for $\multimap$ -left asserts existence of a message $\mathbf{msg}(\mathbf{send}_{x_\beta} y_\alpha)$ and of subtrees $\mathcal{T}_1$ and $\mathcal{T}_2$ in $\mathcal{D}_1$ and $\mathcal{D}_2$, resp. The subtrees $\mathcal{T}_1$ and $\mathcal{T}_2$ are rooted at the sent channel x_β and will be transferred (and thus lost) together with the sent channel. The clause comprises two invocations of the term relation, $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}[\Delta' \Vdash x_\beta.A]$, asserting that future observations to be made along the sent channel x_β are related, and $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta'', y_{\alpha+1}.B \Vdash K]$, asserting that

- (1) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\cdot \Vdash y_\alpha : 1]_{y_\alpha}^{m+1}$ iff $\mathcal{D}_1 = \mathbf{msg}(\mathbf{close} \ y_\alpha)$ and $\mathcal{D}_2 = \mathbf{msg}(\mathbf{close} \ y_\alpha)$
- (2) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta \Vdash y_\alpha : \oplus \{\ell : A_\ell\}_{\ell \in I}]_{y_\alpha}^{m+1}$ iff $\exists k_1, k_2 \in I. k_1 = k_2$ and
 $\mathcal{D}_1 = \mathcal{D}'_1 \mathbf{msg}(y_\alpha.k_1)$ and
 $\mathcal{D}_2 = \mathcal{D}'_2 \mathbf{msg}(y_\alpha.k_2)$ and
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta \Vdash y_{\alpha+1} : A_{k_1}]^m$
- (3) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta \Vdash y_\alpha : \& \{\ell : A_\ell\}_{\ell \in I}]_{y_\alpha}^{m+1}$ iff $\forall k_1, k_2 \in I. \text{ if } k_1 = k_2 \text{ then}$
 $(\mathcal{D}_1 \mathbf{msg}(y_\alpha.k_1); \mathcal{D}_2 \mathbf{msg}(y_\alpha.k_2)) \in$
 $\mathcal{E}[\Delta \Vdash y_{\alpha+1} : A_{k_1}]^m$
- (4) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta', \Delta'' \Vdash y_\alpha : A \otimes B]_{y_\alpha}^{m+1}$ iff $\exists x_\beta \text{ s.t.}$
 $\mathcal{D}_1 = \mathcal{D}'_1 \mathcal{T}_1 \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha)$ and
 $\mathcal{D}_2 = \mathcal{D}'_2 \mathcal{T}_2 \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha)$ and
 $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}[\Delta'' \Vdash x_\beta : A]^m$ and
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta' \Vdash y_{\alpha+1} : B]^m$
- (5) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta \Vdash y_\alpha : A \multimap B]_{y_\alpha}^{m+1}$ iff $\forall x_\beta \notin \text{dom}(\Delta, y_\alpha : A \multimap B).$
 $(\mathcal{D}_1 \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha); \mathcal{D}_2 \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha)) \in$
 $\mathcal{E}[\Delta, x_\beta : A \Vdash y_{\alpha+1} : B]^m$
- (6) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta, y_\alpha : 1 \Vdash K]_{y_\alpha}^{m+1}$ iff $(\mathbf{msg}(\mathbf{close} \ y_\alpha) \mathcal{D}_1; \mathbf{msg}(\mathbf{close} \ y_\alpha) \mathcal{D}_2) \in$
 $\mathcal{E}[\Delta \Vdash K]^m$
- (7) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta, y_\alpha : \oplus \{\ell : A_\ell\}_{\ell \in I} \Vdash K]_{y_\alpha}^{m+1}$ iff $\forall k_1, k_2 \in I. \text{ if } k_1 = k_2 \text{ then}$
 $(\mathbf{msg}(y_\alpha.k_1) \mathcal{D}_1; \mathbf{msg}(y_\alpha.k_2) \mathcal{D}_2) \in$
 $\mathcal{E}[\Delta, y_{\alpha+1} : A_{k_1} \Vdash K]^m$
- (8) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta, y_\alpha : \& \{\ell : A_\ell\}_{\ell \in I} \Vdash K]_{y_\alpha}^{m+1}$ iff $\exists k_1, k_2 \in I. k_1 = k_2$ and
 $\mathcal{D}_1 = \mathbf{msg}(y_\alpha.k_1) \mathcal{D}'_1$ and
 $\mathcal{D}_2 = \mathbf{msg}(y_\alpha.k_2) \mathcal{D}'_2$ and
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta, y_{\alpha+1} : A_{k_1} \Vdash K]^m$
- (9) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta, y_\alpha : A \otimes B \Vdash K]_{y_\alpha}^{m+1}$ iff $\forall x_\beta \notin \text{dom}(\Delta, y_\alpha : A \otimes B, K).$
 $(\mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha) \mathcal{D}_1; \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha) \mathcal{D}_2) \in$
 $\mathcal{E}[\Delta, x_\beta : A, y_{\alpha+1} : B \Vdash K]^m$
- (10) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta', \Delta'', y_\alpha : A \multimap B \Vdash K]_{y_\alpha}^{m+1}$ iff $\exists x_\beta \text{ s.t.}$
 $\mathcal{D}_1 = \mathcal{T}_1 \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha) \mathcal{D}'_1$ and
 $\mathcal{D}_2 = \mathcal{T}_2 \mathbf{msg}(\mathbf{send} \ x_\beta \ y_\alpha) \mathcal{D}'_2$ and
 $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}[\Delta' \Vdash x_\beta : A]^m$ and
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta'', y_{\alpha+1} : B \Vdash K]^m$
- (11) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash K]^{m+1}$ iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathbf{Tree}(\Delta \Vdash K)$ and
 $\forall \Upsilon_1, \Theta_1, \mathcal{D}'_1. \text{ if } \mathcal{D}_1 \mapsto^{*r_1 : \Theta_1} \mathcal{D}'_1 \text{ then}$
 $\exists \Upsilon_2, \mathcal{D}'_2 \text{ such that } \mathcal{D}_2 \mapsto^{*r_2} \mathcal{D}'_2, \text{ and } \Upsilon_1 \subseteq \Upsilon_2 \text{ and}$
 $\forall y_\alpha \in \mathbf{Out}(\Delta \Vdash K).$
 $\text{ if } y_\alpha \in \Upsilon_1. \text{ then } (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\Delta \Vdash K]_{y_\alpha}^{m+1} \text{ and}$
 $\forall y_\alpha \in \mathbf{In}(\Delta \Vdash K).$
 $\text{ if } y_\alpha \in \Theta_1. \text{ then } (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\Delta \Vdash K]_{y_\alpha}^{m+1}$
- (12) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash K]^0$ iff $(\mathcal{D}_1; \mathcal{D}_2) \in \mathbf{Tree}(\Delta \Vdash K)$

Fig. 5: Recursive session logical relation (RSLR) for ILLST. (Clauses (1)-(10) are only defined for well-typed configurations, a condition we elided for concision.)

the continuations $\mathcal{D}_1''$ and $\mathcal{D}_2''$ are related.³ The channel endpoint x_β is existentially quantified and occurs either bound or free in both $\mathcal{D}_1$ and $\mathcal{D}_2$. Conversely, clause (5) in Fig. 5 for $\multimap$ -right assumes receipt of a message $\mathbf{msg}(\mathbf{send} x_\beta y_\alpha)$ that carries a universally quantified free channel x_β , different from the free channels available in the pre-state ($\forall x_\beta \notin \text{dom}(\Delta, y_\alpha : A \multimap B)$). The sequent is enlarged with the received channel x_β for the invocation of the term relation $(\mathcal{D}_1 \mathbf{msg}(\mathbf{send} x_\beta y_\alpha); \mathcal{D}_2 \mathbf{msg}(\mathbf{send} x_\beta y_\alpha)) \in \mathcal{E}[\Delta, x_\beta : A \vdash y_{\alpha+1} : B]$.

The *duality* of ILLST, established by “cutting” the left and right rule of a connective, translates equally to our logical relation, allowing us to compose related configurations in parallel, such that the assumptions made by one pair of configurations are discharged by the other. We prove this result in § 3.2 and exploit it to connect our development to biorthogonality in § 3.5.

Syntactic well-typedness. The value and term interpretations of our logical relation relate pairs of process configurations only if both are syntactically well-typed. The predicate $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}(\Delta \Vdash K)$ stipulates this requirement and is defined as $\mathcal{D}_1 \in \text{Tree}(\Delta \Vdash K)$ and $\mathcal{D}_2 \in \text{Tree}(\Delta \Vdash K)$, with $\mathcal{D}_i \in \text{Tree}(\Delta \Vdash K)$ defined as $\Delta \Vdash \mathcal{D}_i :: K$ (see [5] for details). For concision, we elide appeals to the $\text{Tree}()$ predicate in the value interpretation (clauses (1)-(10) in Fig. 5). Syntactic typing ensures that configurations form *rooted trees*, a property our proof of semantic cut (Lem. 2 in § 3.2) relies upon.

Well-foundedness. In the presence of general recursive types, it is no longer sound to define the logical relation by multiset induction over the structure of types in its sequent $\Delta \Vdash K$. To restore well-foundedness a measure like step-indexing [2, 4] can be employed. Step-indexing is typically tied to the number of type unfolding or computation steps by which one of the two programs is bound. A more natural and symmetric measure for our setting is the number of *observations* that can be made along the free channels $\Delta \Vdash K$ of the logical relation. We thus bound the value and term interpretation of our logical relation by this number and attach the resulting *observation index* as the superscript m to the sequent $\Delta \Vdash K$, yielding clauses of the form

$$(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}[\Delta \Vdash K]^{m+1} \text{ iff } \dots (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}[\Delta' \Vdash K']^m$$

The observation index gets decremented whenever the value interpretation “observes” a message exchange. If $m = 0$, well-typed configurations are trivially related, as expressed by clause (12) in Fig. 5.

Non-termination and nondeterminism. It is now time to take a closer look at the definition of the term interpretation of our logical relation. Its definition is challenged by the possibility of divergence and nondeterminism of scheduling. For the former, a progress-sensitive statement of equivalence demands that the

³ We will momentarily explain the superscript and subscripts of these invocations.

relation relates a divergent program only with another diverging one. For the latter, the term interpretation has to account for the possibility that two configurations may not *simultaneously* be ready to send or receive along a free channel in $\Delta \Vdash K$. To address these challenges, the term interpretation is phrased as follows (we are repeating clause (11) in Fig. 5):

$$\begin{aligned}
& (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash K]^{m+1} \text{ iff} \\
& (\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}(\Delta \Vdash K) \text{ and } \forall \Upsilon_1, \Theta_1, \mathcal{D}'_1. \text{if } \mathcal{D}_1 \mapsto^{*\Upsilon_1, \Theta_1} \mathcal{D}'_1 \text{ then} \\
& \quad \exists \Upsilon_2, \mathcal{D}'_2 \text{ such that } \mathcal{D}_2 \mapsto^{*\Upsilon_2} \mathcal{D}'_2 \text{ and } \Upsilon_1 \subseteq \Upsilon_2 \text{ and} \\
& \quad \forall y_\alpha \in \text{Out}(\Delta \Vdash K). \text{if } y_\alpha \in \Upsilon_1. \text{then } (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\Delta \Vdash K]^{m+1}_{\cdot; y_\alpha} \\
& \quad \text{and } \forall y_\alpha \in \text{In}(\Delta \Vdash K). \text{if } y_\alpha \in \Theta_1. \text{then } (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\Delta \Vdash K]^{m+1}_{y_\alpha; \cdot}
\end{aligned}$$

The term interpretation uses the transition $\mathcal{D}_1 \mapsto^{*\Upsilon_1, \Theta_1} \mathcal{D}'_1$, amounting to the iterated application of the rewriting rules defined in Fig. 4. The star expresses that zero to multiple steps can be taken. The superscript Υ_1 denotes the set of channels in $\Delta \Vdash K$ for which there exist messages in $\mathcal{D}'_1$ to be sent along these channels. The superscript Θ_1 denotes set of channels in $\Delta \Vdash K$ for which there exist processes in $\mathcal{D}'_1$ waiting to receive along these channels.

To ensure progress-sensitivity, the term interpretation asserts, that, whenever $\mathcal{D}_1$ can step, so can $\mathcal{D}_2$. This correspondence is expressed by the condition $\Upsilon_1 \subseteq \Upsilon_2$, ensuring that the messages ready to be sent to the outside world in $\mathcal{D}'_2$ are at least the ones ready to be sent in $\mathcal{D}'_1$. An analogous condition for incoming messages is omitted due to asynchrony of the dynamics where the receipt of a message is not observable. The existentially quantified $\mathcal{D}'_2$ is justified by nondeterminism of scheduling, allowing the term interpretation to choose an appropriate way of stepping $\mathcal{D}_2$ to catch up with $\mathcal{D}'_1$.

The set $\text{Out}(\Delta \Vdash K)$ includes all channels whose types in Δ, K indicate a send. This includes channels in Υ_1 for which the messages are ready to be sent. Similarly, the set $\text{In}(\Delta \Vdash K)$ includes all channels whose types in Δ, K indicate a receive, including those in Θ_1 for which the processes are ready to receive.

To resolve the issue of simultaneity, the term interpretation makes use of two *focus* channels ranging over the sets Υ_1 and Θ_1 . These are added as a subscript $_; _$ to the value interpretation, where $\cdot; y_\alpha$ indicates that $y_\alpha \in \Upsilon_1$ and $y_\alpha; \cdot$ that $y_\alpha \in \Theta_1$. The term interpretation invokes the value interpretation for all the channels in Υ_1 and Θ_1 , thus ensuring that any messages ready to be sent in $\mathcal{D}'_1$ and $\mathcal{D}'_2$ and any processes waiting to receive in $\mathcal{D}'_1$ will be “observed” by the value interpretation. Non-productive configurations are trivially accommodated by the term interpretation, by allowing the sets Υ_1 and Θ_1 to be empty.

Example. We revisit the **Verifier** example (Fig. 2) to showcase the design of our logical relation. Consider the configuration $\mathcal{D} = \text{proc}(x_0, [x_0/x]\text{Verifier})$, in which the channel x_0 instantiates the channel variable x in the **Verifier** process. We outline the first few steps required to establish that $\mathcal{D}$ is self-related at observation index 3, i.e., $(\mathcal{D}; \mathcal{D}) \in \mathcal{E}[\cdot \Vdash x_0.\text{ver}]^3$. By Fig. 4, we observe that $\mathcal{D}$ cannot take any internal steps, that no outgoing message exists in the configuration, and that the only channel on which $\mathcal{D}$ is ready to receive is x_0 . Hence, the

only possible transition for $\mathcal{D}$ is $\mathcal{D} \mapsto^{0_{\emptyset, \{x_0\}}} \mathcal{D}$. It follows that the only value relation invoked by this term interpretation is $(\mathcal{D}; \mathcal{D}) \in \mathcal{V}[\![\cdot \Vdash x_0:\text{ver}]\!]_{x_0}^3$, which we can rewrite as $(\mathcal{D}; \mathcal{D}) \in \mathcal{V}[\![\cdot \Vdash x_0:\text{pin} \multimap \oplus\{\text{succ}:\text{pin} \otimes \text{ver}, \text{fail}:\text{pin} \otimes \text{ver}\}]\!]_{x_0}^3$ by unfolding the type ver ; the latter now matches clause (5) in Fig. 5. To establish this value relation, it remains to show that for an arbitrary $u_\beta \neq x_\alpha$, we can derive $(\mathcal{D}\text{msg}(\text{send } u_\beta x_0); \mathcal{D}\text{msg}(\text{send } u_\beta x_0)) \in \mathcal{E}[u_\beta:\text{pin} \Vdash x_0: \oplus\{\text{succ}:\text{pin} \otimes \text{ver}, \text{fail}:\text{pin} \otimes \text{ver}\}]^2$, which again invokes the term interpretation via clause (11). The remaining steps follow similarly.

A reader may wonder why the logical relation seems oblivious to recursive type unfoldings. This is because the relation is indexed by the number of external observations and is therefore invariant under such unfoldings. In particular, for the example above, $(\mathcal{D}; \mathcal{D}) \in \mathcal{E}[\![\Delta \Vdash x_0:\text{ver}]\!]^3$ and $(\mathcal{D}; \mathcal{D}) \in \mathcal{E}[\![\Delta \Vdash x_0:\text{pin} \multimap \oplus\{\text{succ}:\text{pin} \otimes \text{ver}, \text{fail}:\text{pin} \otimes \text{ver}\}]\!]^3$, are equivalent: the value interpretation with focus on channel x_0 is only triggered when the process is waiting to receive a message along x_0 . After consumption of the message, the recursive invocation of the term interpretation in clause (5) will then happen at index 2.

3.2 Semantic cut: closure under parallel composition

The use of an observation index rather than a step index by the RSLR is fundamental in proving various metatheoretic properties in the remainder of this section. It allows us to keep the observation index *constant* in the term interpretation, while stepping the configurations internally, permitting disagreement in the number of internal messages exchanged. As a result, internal steps cannot depend on the observation index, facilitating proof of the following lemma:

Lemma 1 (Moving existential over universal quantifier).

If

$$\begin{aligned} & \forall m. \forall \Upsilon_1, \Theta_1, \mathcal{D}'_1. \text{ if } \mathcal{D}_1 \mapsto^{*r_1: \Theta_1} \mathcal{D}'_1, \text{ then} \\ & \exists \Upsilon_2, \mathcal{D}'_2 \text{ such that } \mathcal{D}_2 \mapsto^{*r_2} \mathcal{D}'_2 \text{ and } \Upsilon_1 \subseteq \Upsilon_2 \text{ and} \\ & \quad \forall x_\alpha \in \mathbf{Out}(\Delta \Vdash K). \text{ if } x_\alpha \in \Upsilon_1. \text{ then} \\ & \quad \quad (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\![\Delta \Vdash K]\!]_{x_\alpha}^{m+1} \text{ and} \\ & \quad \forall x_\alpha \in \mathbf{In}(\Delta \Vdash K). \text{ if } x_\alpha \in \Theta_1. \text{ then} \\ & \quad \quad (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\![\Delta \Vdash K]\!]_{x_\alpha}^{m+1}, \end{aligned}$$

then

$$\begin{aligned} & \forall \Upsilon_1, \Theta_1, \mathcal{D}'_1. \text{ if } \mathcal{D}_1 \mapsto^{*r_1: \Theta_1} \mathcal{D}'_1, \text{ then} \\ & \exists \Upsilon_2, \mathcal{D}'_2 \text{ such that } \mathcal{D}_2 \mapsto^{*r_2} \mathcal{D}'_2 \text{ and } \Upsilon_1 \subseteq \Upsilon_2 \text{ and} \\ & \quad \forall x_\alpha \in \mathbf{Out}(\Delta \Vdash K). \text{ if } x_\alpha \in \Upsilon_1. \text{ then} \\ & \quad \quad \forall m. (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\![\Delta \Vdash K]\!]_{x_\alpha}^{m+1} \text{ and} \\ & \quad \forall x_\alpha \in \mathbf{In}(\Delta \Vdash K). \text{ if } x_\alpha \in \Theta_1. \text{ then} \\ & \quad \quad \forall m. (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}[\![\Delta \Vdash K]\!]_{x_\alpha}^{m+1}. \end{aligned}$$

Proof. The full proof is available in [5]. The proof relies on confluence and backwards closure.

With this lemma in hand, we can now prove that the logical relation is closed under parallel composition.

Lemma 2 (Semantic cut). $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta, u_\alpha : T \Vdash K]^m$ iff for all $\mathcal{T}_1$ and $\mathcal{T}_2$ if $\forall m. (\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}[\Delta' \Vdash u_\alpha : T]^m$ then $\forall k. (\mathcal{T}_1 \mathcal{D}_1; \mathcal{T}_2 \mathcal{D}_2) \in \mathcal{E}[\Delta', \Delta \Vdash K]^k$.

Proof. The full proof is available in [5]. The proof relies on Lem. 1.

The semantic cut lemma is analogous to the syntactic cut-elimination theorem, with T serving as the analog of the cut formula. Consider the left-to-right direction. For internal steps in $\mathcal{T}_1 \mathcal{D}_1$ invoked by the term interpretation that do not involve the channel $u_\alpha : T$, we can perform corresponding internal steps in $\mathcal{T}_1$ or $\mathcal{D}_1$. Using the fact that $\mathcal{T}_1$ and $\mathcal{D}_1$ are related to $\mathcal{T}_2$ and $\mathcal{D}_2$, resp., we can construct the next step(s) of $\mathcal{T}_2 \mathcal{D}_2$ as required by the term interpretation. For internal steps in $\mathcal{T}_1 \mathcal{D}_1$ along the channel $u_\alpha : T$, we use the fact that $\mathcal{T}_1$ is ready to send a message along u_α and $\mathcal{D}_1$ is ready to receive along u_α (or vice versa). Here, we take advantage of the duality in the value interpretation for the provider and client of each connective to show that the assertions made by the sender are sufficient to establish the assumptions of the receiver. With this, we invoke the definition of value interpretations to transition to a new term interpretation involving subformula(s) of type T in the sequent and apply the inductive hypothesis. This step is analogous to internal cut reductions in the cut-elimination proof. For any communication in $\mathcal{T}_1 \mathcal{D}_1$ along its external channel, we use the fact that either $\mathcal{T}_1$ or $\mathcal{D}_1$ can perform the same communication. Using their respective value interpretations and the inductive hypothesis, we establish the value interpretation for $\mathcal{T}_1 \mathcal{D}_1$ and $\mathcal{T}_2 \mathcal{D}_2$. This step corresponds to external cut reductions in the cut-elimination proof.

3.3 Logical equivalence

To prove that the RSLR induces an equivalence relation, we introduce the notation $\Delta \Vdash \mathcal{D}_1 :: K \equiv \Delta \Vdash \mathcal{D}_2 :: K$, defined below, where $(\mathcal{C}_1; \mathcal{C}_2) \in \text{Forest}(\Delta' \Vdash \Delta)$ stands for $\Delta' \Vdash \mathcal{C}_1 :: \Delta$ and $\Delta' \Vdash \mathcal{C}_2 :: \Delta$:

Definition 1 (Logical equivalence).

- We define the relation $\Delta \Vdash \mathcal{D}_1 :: K \equiv \Delta \Vdash \mathcal{D}_2 :: K$ as $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}(\Delta \Vdash K)$ and $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash K]^m$ and $\forall m. (\mathcal{D}_2; \mathcal{D}_1) \in \mathcal{E}[\Delta \Vdash K]^m$.
- We define the relation $\Delta \vdash \mathcal{C}_1[] \mathcal{F}_1 \vdash K \equiv \Delta \vdash \mathcal{C}_2[] \mathcal{F}_2 \vdash K$ as (i) $\exists K'$ such that $K \Vdash \mathcal{F}_1 :: K' \equiv K \Vdash \mathcal{F}_2 :: K'$ and (ii) $\exists \Delta'$ such that $(\mathcal{C}_1; \mathcal{C}_2) \in \text{Forest}(\Delta' \Vdash \Delta)$ and $\forall \mathcal{T}_1 \in \mathcal{C}_1$, and $\forall \mathcal{T}_2 \in \mathcal{C}_2$ with $(\mathcal{T}_1; \mathcal{T}_2) \in \text{Tree}(\Delta'_1 \Vdash K'')$, we have $\Delta'_1 \Vdash \mathcal{T}_1 :: K'' \equiv \Delta'_1 \Vdash \mathcal{T}_2 :: K''$.

Next, we show that the relation $\equiv$ is reflexive, symmetric, and transitive.

Lemma 3 (Reflexivity). For all configurations $\Delta \Vdash \mathcal{D} :: x_\alpha : T$, we have $(\Delta \Vdash \mathcal{D} :: x_\alpha : T) \equiv (\Delta \Vdash \mathcal{D} :: x_\alpha : T)$.

Proof. The full proof is available in [5].

Lemma 4 (Symmetry). For all configurations $\mathcal{D}_1$ and $\mathcal{D}_2$, we have $(\Delta \Vdash \mathcal{D}_1 :: x_\alpha : T) \equiv (\Delta \Vdash \mathcal{D}_2 :: x_\alpha : T)$, iff $(\Delta \Vdash \mathcal{D}_2 :: x_\alpha : T) \equiv (\Delta \Vdash \mathcal{D}_1 :: x_\alpha : T)$,

Proof. The proof is straightforward by Def. 1.

Lemma 5 (Transitivity). *For all configurations $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$, we have*

$$\begin{aligned} & \text{if } (\Delta \Vdash \mathcal{D}_1 :: x_\alpha:T) \equiv (\Delta \Vdash \mathcal{D}_2 :: x_\alpha:T), \text{ and} \\ & (\Delta \Vdash \mathcal{D}_2 :: x_\alpha:T) \equiv (\Delta \Vdash \mathcal{D}_3 :: x_\alpha:T) \\ & \text{then } (\Delta \Vdash \mathcal{D}_1 :: x_\alpha:T) \equiv (\Delta \Vdash \mathcal{D}_3 :: x_\alpha:T). \end{aligned}$$

Proof. The full proof is available in [5].

3.4 Adequacy

To show that the RSLR is adequate, we prove that configurations related by the logical relation are bisimilar and vice versa. To facilitate this proof, we first give the definition of weak *asynchronous bisimilarity* [57] (Def. 2). The definition relies on a standard labeled transition system displayed in Fig. 6, which rephrases our dynamics in Fig. 4. We define *weak* transitions as

- (1) $\Rightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$,
- (2) $\xRightarrow{\alpha}$ is $\Rightarrow \xrightarrow{\alpha}$.

Definition 2 (Weak asynchronous bisimilarity). *Asynchronous bisimilarity, written $\mathcal{D}_1 \approx_a \mathcal{D}_2$, is the largest symmetric relation such that whenever $\mathcal{D}_1 \approx_a \mathcal{D}_2$, we have*

- (τ – step) *if $\mathcal{D}_1 \xrightarrow{\tau} \mathcal{D}'_1$ then $\exists \mathcal{D}'_2. \mathcal{D}_2 \xRightarrow{\tau} \mathcal{D}'_2$ and $\mathcal{D}'_1 \approx_a \mathcal{D}'_2$,*
- (output) *if $\mathcal{D}_1 \xrightarrow{\overline{x}_\alpha q} \mathcal{D}'_1$ then $\exists \mathcal{D}'_2. \mathcal{D}_2 \xRightarrow{\overline{x}_\alpha q} \mathcal{D}'_2$ and $\mathcal{D}'_1 \approx_a \mathcal{D}'_2$.*
- (left input) *for all $q \notin \text{fn}(\mathcal{D}_1)$, if $\mathcal{D}_1 \xrightarrow{\mathbf{L} x_\alpha q} \mathcal{D}'_1$ then $\exists \mathcal{D}'_2. \mathcal{D}_2 \xRightarrow{\tau} \mathcal{D}'_2$ and $\mathcal{D}'_1 \approx_a \mathbf{msg}(x_\alpha.q)\mathcal{D}'_2$,*
- (right input) *for all $q \notin \text{fn}(\mathcal{D}_1)$, if $\mathcal{D}_1 \xrightarrow{\mathbf{R} x_\alpha q} \mathcal{D}'_1$ then $\exists \mathcal{D}'_2. \mathcal{D}_2 \xRightarrow{\tau} \mathcal{D}'_2$ and $\mathcal{D}'_1 \approx_a \mathcal{D}'_2 \mathbf{msg}(x_\alpha.q)$.*

where $\mathbf{msg}(x_\alpha.q)$ is defined as $\mathbf{msg}(\text{close } x_\alpha)$ if $q = \text{close}$, $\mathbf{msg}(x_\alpha.k)$ if $q = k$, and $\mathbf{msg}(\text{send } z_\delta x_\alpha)$ if $q = z_\delta$. Here, $\text{fn}(\mathcal{D}_1)$ is the set of free channels in the configuration $\mathcal{D}_1$.

Our adequacy theorem (Lem. 6) shows that RSLRs are sound and complete with regard to asynchronous bisimilarity.

Lemma 6 (Adequacy). *For all $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}(\Delta \Vdash K)$, we have $\mathcal{D}_1 \approx_a \mathcal{D}_2$ iff $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}[\Delta \Vdash K]^m$ and $\forall m. (\mathcal{D}_2; \mathcal{D}_1) \in \mathcal{E}[\Delta \Vdash K]^m$.*

Proof. The full proof is available in [5].

Internal transition $\xrightarrow{\tau}$ defined as:

$$\mathcal{D}_1 \xrightarrow{\tau} \mathcal{D}'_1 \text{ iff } \mathcal{D}_1 \mapsto \mathcal{D}'_1$$

Actions $\xrightarrow{\overline{y}_\alpha q}$, $\xrightarrow{\mathbf{L} y_\alpha q}$ and $\xrightarrow{\mathbf{R} y_\alpha q}$ defined as below when $y \in \mathbf{fn}(\mathcal{D}_1)$:

- (1) $\mathcal{D}_1 \text{msg}(\text{close } y_\alpha) \mathcal{D}_2 \xrightarrow{\overline{y}_\alpha \text{close}}$
 $\mathcal{D}_1 \mathcal{D}_2$
- (2) $\mathcal{D}_1 \text{msg}(y_\alpha.k) \mathcal{D}_2 \xrightarrow{\overline{y}_\alpha k}$
 $\mathcal{D}_1 \mathcal{D}_2$
- (3) $\mathcal{D}_1 \text{msg}(\text{send } x_\beta y_\alpha) \mathcal{D}_2 \xrightarrow{\overline{y}_\alpha x_\beta}$
 $\mathcal{D}_1 \mathcal{D}_2$
- (4) $\mathcal{D}_1 \text{proc}(z_\delta, \text{wait } y_\alpha; P) \mathcal{D}_2 \xrightarrow{\mathbf{L} y_\alpha \text{close}}$
 $\text{msg}(\text{close } y_\alpha) \mathcal{D}_1 \text{proc}(z_\delta, \text{wait } y_\alpha; P) \mathcal{D}_2$
- (5) $\mathcal{D}_1 \text{proc}(z_\delta, \text{case } y_\alpha (\ell \Rightarrow P_\ell)_{\ell \in I}) \mathcal{D}_2 \xrightarrow{\mathbf{L} y_\alpha k}$
 $\text{msg}(y_\alpha.k) \mathcal{D}_1 \text{proc}(z_\delta, \text{case } y_\alpha (\ell \Rightarrow P_\ell)_{\ell \in I}) \mathcal{D}_2$
- (6) $\mathcal{D}_1 \text{proc}(z_\delta, w \leftarrow \text{recv } y_\alpha) \mathcal{D}_2 \xrightarrow{\mathbf{L} y_\alpha x_\beta}$
 $\text{msg}(\text{send } x_\beta y_\alpha) \mathcal{D}_1 \text{proc}(z_\delta, w \leftarrow \text{recv } y_\alpha) \mathcal{D}_2$
- (7) $\mathcal{D}_1 \text{proc}(y_\alpha, \text{case } y_\alpha (\ell \Rightarrow P_\ell)_{\ell \in I}) \mathcal{D}_2 \xrightarrow{\mathbf{R} y_\alpha k}$
 $\mathcal{D}_1 \text{proc}(y_\alpha, \text{case } y_\alpha (\ell \Rightarrow P_\ell)_{\ell \in I}) \mathcal{D}_2 \text{msg}(y_\alpha.k)$
- (8) $\mathcal{D}_1 \text{proc}(y_\alpha, w \leftarrow \text{recv } y_\alpha) \mathcal{D}_2 \xrightarrow{\mathbf{R} y_\alpha x_\beta}$
 $\mathcal{D}_1 \text{proc}(y_\alpha, w \leftarrow \text{recv } y_\alpha) \mathcal{D}_2 \text{msg}(\text{send } x_\beta y_\alpha)$

Fig. 6: Labeled transition system for SESSION.

3.5 Biorthogonality

Our main result, biorthogonal closure of $\equiv$ (Thm. 1), relies upon various definitions, including an *orthogonality* operation $\top$ [9, 43, 51] (Def. 5), which we give next. We follow the development of Pitts [51].

We first define the program relations $P\text{Rel}(\Delta \Vdash K)$ and environment relations $E\text{Rel}(\Delta \Vdash K)$ (Def. 4), reliant upon a session-typed environment (Def. 3):

Definition 3 (Session-typed environment). *A session-typed environment with the sequent $\Delta \Vdash K$, is of the form $\mathcal{C}[\]\mathcal{F}$, such that for some Δ' and K' , we have $\Delta' \Vdash \mathcal{C} :: \Delta$ and $K \Vdash \mathcal{F} :: K'$.*

Definition 4 (Program- and environment- relations). *A session program-relation is a binary relation between session-typed programs, i.e., open configurations of the form $\Delta \Vdash \mathcal{D} :: K$. Given the sequent $\Delta \Vdash K$, we write $P\text{Rel}(\Delta \Vdash K)$ for the set of all program relations that relate programs $\Delta \Vdash \mathcal{D} :: K$.*

A session environment-relation is a binary relation between session-typed environments. Given the sequent $\Delta \Vdash K$, we write $E\text{Rel}(\Delta \Vdash K)$ for the set of all environment relations that relate environments $C[\]\mathcal{F}$ with the sequent $\Delta \Vdash K$.

We can now define the orthogonality operation $\top$ [43, 51].

Definition 5 (The $(_)^\top$ operation). Given sequent $\Delta \Vdash K$ and program relation $r \in P\text{Rel}(\Delta \Vdash K)$, we define $r^\top \in E\text{Rel}(\Delta \Vdash K)$ by

$$\begin{aligned} (C_1[\]\mathcal{F}_1, C_2[\]\mathcal{F}_2) &\in r^\top \text{ iff} \\ \forall (\mathcal{D}_1, \mathcal{D}_2) &\in r. (C_1\mathcal{D}_1\mathcal{F}_1 \approx_a C_2\mathcal{D}_2\mathcal{F}_2), \end{aligned}$$

and given $s \in E\text{Rel}(\Delta \Vdash K)$, we define $s^\top \in P\text{Rel}(\Delta \Vdash K)$ by

$$\begin{aligned} (\mathcal{D}_1, \mathcal{D}_2) &\in s^\top \text{ iff} \\ \forall (C_1[\]\mathcal{F}_1, C_2[\]\mathcal{F}_2) &\in s. (C_1\mathcal{D}_1\mathcal{F}_1 \approx_a C_2\mathcal{D}_2\mathcal{F}_2). \end{aligned}$$

By definition, the $\top$ operation is inflationary and idempotent [51].

Finally, we state our main result that the logical equivalence relation $\equiv$ induced by our RSLR (Def. 1) is sound and complete with regard to closure of weak bisimilarity under parallel composition.

Theorem 1 ($\top\top$ -closure). Consider $(\mathcal{D}_1, \mathcal{D}_2) \in \text{Tree}(\Delta \Vdash K)$, we have

$$\begin{aligned} (\Delta \Vdash \mathcal{D}_1 :: K) &\equiv (\Delta \Vdash \mathcal{D}_2 :: K) \text{ iff} \\ \forall C_1, C_2, \mathcal{F}_1, \mathcal{F}_2. &\text{ if } (\Delta \dashv C_1[\]\mathcal{F}_1 \dashv K) \equiv (\Delta \dashv C_2[\]\mathcal{F}_2 \dashv K) \\ &\text{ then } C_1\mathcal{D}_1\mathcal{F}_1 \approx_a C_2\mathcal{D}_2\mathcal{F}_2. \end{aligned}$$

Proof. The full proof is available in [5].

The main purpose of Thm. 1 is to validate the logical equivalence induced by our logical relation, ensuring that it has enough discriminatory power (soundness) while being maximally permissive (completeness). To do that, Thm. 1 uses biorthogonal closure. As detailed in [51], to prove that a relation r is $\top\top$ -closed, i.e., $r = r^{\top\top}$, it is enough to show that for some witness relation s , we have $r = s^\top$. Thm. 1 instantiates the program-relation r and environment-relation s (Def. 5) with the logical equivalences (Def. 1) induced by our logical relation: it defines $(\mathcal{D}_1, \mathcal{D}_2) \in r$ as the program relation $(\Delta \Vdash \mathcal{D}_1 :: K) \equiv (\Delta \Vdash \mathcal{D}_2 :: K)$, and shows that if we define the witness relation $(C_1[\]\mathcal{F}_1, C_2[\]\mathcal{F}_2) \in s$ as the environment relation $(\Delta \dashv C_1[\]\mathcal{F}_1 \dashv K) \equiv (\Delta \dashv C_2[\]\mathcal{F}_2 \dashv K)$, then we can prove $r = s^\top$. In particular, the theorem proves that related programs cannot be discriminated by related environments ($r \subseteq s^\top$) and that programs that cannot be discriminated by related environments are related ($s^\top \subseteq r$). Thm. 1 is stated in terms of the definitions of the relations r and s , and Def. 5, i.e., the left side is equivalent to $(\mathcal{D}_1, \mathcal{D}_2) \in r$, by the definition of r , and the right side is equivalent to $(\mathcal{D}_1, \mathcal{D}_2) \in s^\top$ by Def. 5 and s .

Thm. 1 thus establishes that our logical relation is compositional and extensional (behavioral). It shows that our logical relation includes exactly the programs whose compositions with related environments are observationally equivalent. Soundness ensures that related environments are not very strong and do not have too much discriminatory power with respect to the observable behavior – related environments cannot distinguish between the related programs. Completeness guarantees that related environments are not very weak and have enough discriminatory power – if related environments cannot distinguish between two programs, the programs are indeed related. This aligns with the assert-assume *pas de deux* described in § 3.1, ensuring that the assumptions made by related programs are exactly those asserted by the related environments and the assertions of the programs are exactly those assumed by the environments.

4 Noninterference

This section applies the RSLR to noninterference, phrased as an equivalence up to the secrecy level ξ of an observer, $\equiv_{\xi}^{\Psi_0}$, given a security lattice Ψ_0 (§ 4.2). The section then illustrates the up-to equivalence $\equiv_{\xi}^{\Psi_0}$ on an example (§ 4.3) and concludes with its metatheoretic properties (§ 4.4), including $\top\top$ -closure.

4.1 Attacker model

Our attacker model is parametric in the secrecy level $\xi \in \Psi_0$ of an attacker and a program P , given prior annotation of the free channels of P with secrecy levels $c \in \Psi_0$. It assumes

- that the attacker knows the source code of P ;
- that the attacker can only observe the messages sent along the free channels of P with secrecy level $c \sqsubseteq \xi$;
- that an attacker cannot measure the passing of time;
- that an attacker is oblivious of channel names;
- a nondeterministic scheduler.

4.2 Up-to equivalence

To prove noninterference, we introduce an *equivalence relation up to* the secrecy level ξ of an observer, $\equiv_{\xi}^{\Psi_0}$, defined in Def. 8 below. This relation is reminiscent of logical equivalence $\equiv$ (Def. 1), but expects free channels to be annotated with secrecy levels and allows ignoring those channels with secrecy level $\sqsubseteq \xi$.

We first define two auxiliary notions: (i) downward projections on context of free channels $\Gamma \Downarrow \xi$ and on the providing free channel $x_{\alpha}:T[c] \Downarrow \xi$ (Def. 6), as well as (ii) the predicates **H-Provider** $^{\xi}$ and **H-Client** $^{\xi}$ (Def. 7). Since channels are now annotated with secrecy levels, we use the metavariables Γ and K^s to range over the linear typing context and providing channel, resp.

Definition 6 (Typing context projections). *Downward projection on security linear contexts Γ and providing channels K^s is defined as follows:*

$$\begin{aligned}
\Gamma, x_\alpha:T[c] \Downarrow \xi &\stackrel{\text{def}}{=} \Gamma \Downarrow \xi, x_\alpha:T[c] && \text{if } c \sqsubseteq \xi \\
\Gamma, x_\alpha:T[c] \Downarrow \xi &\stackrel{\text{def}}{=} \Gamma \Downarrow \xi && \text{if } c \not\sqsubseteq \xi \\
\cdot \Downarrow \xi &\stackrel{\text{def}}{=} \cdot \\
x_\alpha:T[c] \Downarrow \xi &\stackrel{\text{def}}{=} x_\alpha:T[c] && \text{if } c \sqsubseteq \xi \\
x_\alpha:T[c] \Downarrow \xi &\stackrel{\text{def}}{=} _ : 1[\top] && \text{if } c \not\sqsubseteq \xi
\end{aligned}$$

The projections are used to keep only the free channels that are observable to an attacker. We close the nonobservable channels off by composing the configuration with high-confidentiality clients (**H-Client** $^\xi$) and providers (**H-Provider** $^\xi$), which may differ for each configuration. These high-confidentiality clients and providers are connected to the configurations through nonobservable channels, making the messages they exchange with the configurations nonobservable to the attacker as well.

Definition 7 (High provider and High client).

$$\begin{aligned}
&\cdot \in \mathbf{H}\text{-}\mathbf{Provider}^\xi(\cdot) \\
&\mathcal{B} \in \mathbf{H}\text{-}\mathbf{Provider}^\xi(\Gamma, x_\alpha:A[c]) \quad \text{iff} \\
&\quad \text{either } c \not\sqsubseteq \xi \text{ and } \mathcal{B} = \mathcal{B}'\mathcal{T} \text{ and } \mathcal{B}' \in \mathbf{H}\text{-}\mathbf{Provider}^\xi(\Gamma) \text{ and} \\
&\quad \quad \mathcal{T} \in \text{Tree}(\cdot \Vdash x_\alpha:A), \\
&\quad \text{or } c \sqsubseteq \xi \text{ and } \mathcal{B} \in \mathbf{H}\text{-}\mathbf{Provider}^\xi(\Gamma) \\
&\mathcal{T} \in \mathbf{H}\text{-}\mathbf{Client}^\xi(x_\alpha:A[c]) \quad \text{iff} \\
&\quad \text{either } c \not\sqsubseteq \xi \text{ and } \mathcal{T} \in \text{Tree}(x_\alpha:A \Vdash _ : 1), \text{ or} \\
&\quad \text{or } c \sqsubseteq \xi \text{ and } \mathcal{B} = \cdot
\end{aligned}$$

Finally, we can state the relation $\equiv_\xi^{\psi_0}$. Like $\equiv$ (Def. 1), $\equiv_\xi^{\psi_0}$ is phrased in terms of the term interpretation of the RSLR, but restricts observations to those free channels that are observable. The remaining free channels will be composed in parallel with arbitrary ILLST-typed configurations.

Definition 8 (Logical equivalence up to observer level). *We define the relation $(\Gamma_1 \Vdash \mathcal{D}_1 :: x_\alpha:A_1[c_1]) \equiv_\xi^{\psi_0} (\Gamma_2 \Vdash \mathcal{D}_2 :: y_\beta:A_2[c_2])$ as*

$$\begin{aligned}
&\mathcal{D}_1 \in \text{Tree}(|\Gamma_1| \Vdash x_\alpha:A_1) \text{ and } \mathcal{D}_2 \in \text{Tree}(|\Gamma_2| \Vdash y_\beta:A_2) \text{ and} \\
&\Gamma_1 \Downarrow \xi = \Gamma_2 \Downarrow \xi = \Gamma \text{ and } x_\alpha:A_1[c_1] \Downarrow \xi = y_\beta:A_2[c_2] \Downarrow \xi = K^s \text{ and} \\
&\forall \mathcal{B}_1 \in \mathbf{H}\text{-}\mathbf{Provider}^\xi(\Gamma_1). \forall \mathcal{B}_2 \in \mathbf{H}\text{-}\mathbf{Provider}^\xi(\Gamma_2). \\
&\quad \forall \mathcal{T}_1 \in \mathbf{H}\text{-}\mathbf{Client}^\xi(x_\alpha:A_1[c_1]). \forall \mathcal{T}_2 \in \mathbf{H}\text{-}\mathbf{Client}^\xi(y_\beta:A_2[c_2]). \\
&\quad \forall m. (\mathcal{B}_1 \mathcal{D}_1 \mathcal{T}_1, \mathcal{B}_2 \mathcal{D}_2 \mathcal{T}_2) \in \mathcal{E}[|\Gamma| \Vdash |K^s|]^m, \text{ and} \\
&\quad \forall m. (\mathcal{B}_2 \mathcal{D}_2 \mathcal{T}_2, \mathcal{B}_1 \mathcal{D}_1 \mathcal{T}_1) \in \mathcal{E}[|\Gamma| \Vdash |K^s|]^m.
\end{aligned}$$

Def. 8 makes use of an erasure operation $|_|$ to drop the secrecy annotations, to match the sequent of the logical relation whose free channels lack secrecy annotations. As a result, $\equiv_\xi^{\psi_0}$ can be used for configurations that type check using an information flow control (IFC) type system as well as “plain-vanilla” ILLST-typed configurations, facilitating *semantic typing* [15, 42, 62] for progress-sensitive noninterference (PSNI).

4.3 Case study: progress-sensitive noninterference (PSNI)

To illustrate $\equiv_{\xi}^{\Psi_0}$, we instantiate it on a simplistic example, in the spirit of an insecure version of the Verifier encountered earlier (Fig. 2), shown below. Assuming that our security lattice (Ψ_0) is **guest** $\sqsubseteq$ **alice**, and that the attacker level is **guest**, the below process leaks to an attacker y whether authentication was successful or not by either sending the label s or f , resp.

$$\begin{aligned} \text{pin} &= \&\{tok_1:1, tok_2:1\} & \quad \text{bit} = \&\{zero:1, one:1\} \\ y:\text{bit}[\text{guest}] \vdash X :: x:\text{pin}[\text{alice}] &= (\text{case } x (tok_1 \Rightarrow y.zero; \text{wait } y; \text{close } x \\ & \quad | tok_2 \Rightarrow y.one; \text{wait } y; \text{close } x)) \end{aligned}$$

To prove that this process is secure, we would have to show that for any substitution that instantiates free variables x and y with free channels x_α and y_β

$$\begin{aligned} y_\beta : \text{bit}[\text{guest}] \vdash \text{proc}(x_\alpha, X) :: x_\alpha:\text{pin}[\text{alice}] &\equiv_{\text{guest}}^{\Psi_0} \\ y_\beta : \text{bit}[\text{guest}] \vdash \text{proc}(x_\alpha, X) :: x_\alpha:\text{pin}[\text{alice}] & \end{aligned}$$

We compose the process with the following high-secrecy clients $\mathcal{T}_1$ and $\mathcal{T}_2$, sending different tokens along the high-secrecy channel $x_\alpha:\text{pin}[\text{alice}]$,

$$\begin{aligned} \mathcal{T}_1 &= \text{proc}(z_\eta, x_\alpha.tok_1; \text{wait } x_\alpha; \text{close } z_\eta) \\ \mathcal{T}_2 &= \text{proc}(z_\eta, x_\alpha.tok_2; \text{wait } x_\alpha; \text{close } z_\eta), \end{aligned}$$

leaving us left to show that for all m :

$$\begin{aligned} &(\text{proc}(x_\alpha, X) \text{ proc}(z_\eta, x_\alpha.tok_1; \text{wait } x_\alpha; \text{close } z_\eta); \\ & \text{proc}(x_\alpha, X) \text{ proc}(z_\eta, x_\alpha.tok_2; \text{wait } x_\alpha; \text{close } z_\eta)) \in \mathcal{E}[\![y_\beta:\text{bit} \vdash _ : 1]\!]^m \end{aligned}$$

Obviously this does not hold true, confirming that process X is insecure. Consider a variant of X that sends the bit *zero* in either case. This variant would be accepted by our logical relation, as it should because it is secure.

Alternatively, we could employ an *information flow control (IFC)* type system [55, 58, 65] to verify whether process X is secure. Such type systems label observables (e.g., output, locations, channels) with secrecy levels drawn from a given security lattice Ψ_0 to prevent “flows from high to low”. Next, we sketch such an IFC type system as a refinement of **SESSION**. The full refinement type system is included in [5]. It is based on existing work [17, 18], and thus we only summarize its main characteristics and discuss the typing rules relevant for the example.

To type process terms, we use the judgment

$$\Psi; \Gamma \vdash_{\Sigma} P@c :: x : A[d].$$

The new elements, compared to **SESSION**’s judgment (see § 2), is the security lattice Ψ and the possible worlds annotations $@c$ and $[d]$ [8], to denote a process’ running secrecy and maximal secrecy, resp. Γ is a linear typing context, like Δ , but where types now have maximal secrecy annotations. The idea is that a

process' *maximal secrecy* indicates the maximal level of secret information the process may ever obtain, whereas a process' *running secrecy* denotes the highest level of secret information the process has obtained so far.

To constrain the propagation of information, the following invariants are presupposed on the typing judgment $\Psi; \Gamma \vdash_{\Sigma} P@c :: x : A[d]$

- (i) $\forall y:B[d'] \in \Gamma. \Psi \Vdash d' \sqsubseteq d$
- (ii) $\Psi \Vdash c \sqsubseteq d$

ensuring that the maximal secrecy of a child node is capped by the maximal secrecy of its parent and that the running secrecy of a process is less than or equal to its maximal secrecy, resp. Additionally, the type system ensures that

1. no channel of high secrecy is sent along one with lower secrecy; and
2. a process does not send any message along a low-secrecy channel after receipt of high secrecy information.

The first condition prevents direct flows and is enforced by requiring that the maximal secrecy level of a carrier channel and the channels sent over it match. The second condition is met by ensuring that a process' running secrecy is a sound approximation of the level of secret information a process has obtained so far. To this end, the type system increases the running secrecy of a process upon receipt to *at least* the maximal secrecy of the sending process and, correspondingly, guards sends by making sure that the running secrecy of the sending process is *at most* the maximal secrecy of the receiving process.

This working can be seen in the typing rules for external choice:

$$\frac{\Psi; \Gamma \vdash_{\Sigma} Q_k@c :: y:A_k[c] \quad \forall k \in L}{\Psi; \Gamma \vdash_{\Sigma} (\text{case } y^c(\ell \Rightarrow Q_{\ell})_{\ell \in I})@d_1 :: y:\&\{\ell : A_{\ell}\}_{\ell \in L}[c]} \&R$$

$$\frac{\Psi \Vdash d_1 \sqsubseteq c \quad \Psi; \Gamma, x:A_k[c] \vdash_{\Sigma} P@d_1 :: y:C[c'] \quad k \in L}{\Psi; \Gamma, x:\&\{\ell : A_{\ell}\}_{\ell \in I}[c] \vdash_{\Sigma} (x^c.k; P)@d_1 :: y:C[c']} \&L$$

The right rule increases the provider's running secrecy from d_1 to its maximal secrecy c after receipt of the label k . The left rule guards the send by the premise $\Psi \Vdash d_1 \sqsubseteq c$, demanding that the provider's running secrecy d_1 is less than or equal to the maximal secrecy c of the recipient. It is precisely this guard that prevents process **X** to type check using the refinement IFC type system. This guard also prevents the variant of **X**, which sends the bit *zero* in either case, to type check, which is accepted by our logical relation. This outcome is expected because type systems have the benefit of automatic verification, foregoing completeness. Our logical relation thus amounts to a *semantic* logical relation [15, 42, 62] for PSNI, as it only requires configurations to be ILLST-typed, but not IFC-typed. The fact that IFC-typed processes are non-interfering is proved as the *fundamental theorem* of the logical relation, see [5].

4.4 Metatheoretic properties of up-to equivalence

Unsurprisingly, $\equiv_{\xi}^{\Psi_0}$ has analogous properties to $\equiv$, except for reflexivity, which only holds for IFC-typed configurations, as demonstrated by our **X** process in

§ 4.3. We provide a selection of the most important properties of $\equiv_{\xi}^{\Psi_0}$, the complete listing is given in [5].

Definition 9 (Bisimulation with secrecy annotated sequent). For $\mathcal{D}_1 \in \text{Tree}(|\Gamma_1| \Vdash x_{\alpha}:A_1)$, $\mathcal{D}_2 \in \text{Tree}(|\Gamma_2| \Vdash y_{\beta}:A_2)$ we define $\Gamma_1 \Vdash \mathcal{D}_1 :: x_{\alpha}:A_1[c_1] \approx_a^{\xi} \Gamma_2 \Vdash \mathcal{D}_2 :: y_{\beta}:A_2[c_2]$ as

$$\begin{aligned} \Gamma_1 \Downarrow \xi = \Gamma_2, \Downarrow \xi \text{ and } y_{\beta}:A_2[c_2] \Downarrow \xi = x_{\alpha}:A_1[c_1] \Downarrow \xi \text{ and} \\ \forall \mathcal{B}_1 \in \text{H-Provider}^{\xi}(\Gamma_1). \forall \mathcal{B}_2 \in \text{H-Provider}^{\xi}(\Gamma_2). \\ \forall \mathcal{T}_1 \in \text{H-Client}^{\xi}(x_{\alpha}:A_1[c_1]). \forall \mathcal{T}_2 \in \text{H-Client}^{\xi}(y_{\beta}:A_2[c_2]). \\ \mathcal{B}_1 \mathcal{D}_1 \mathcal{T}_1 \approx_a \mathcal{B}_2 \mathcal{D}_2 \mathcal{T}_2. \end{aligned}$$

Lemma 7 (Adequacy). For all $\mathcal{D}_1 \in \text{Tree}(|\Gamma_1| \Vdash x_{\alpha}:A_1)$ and $\mathcal{D}_2 \in \text{Tree}(|\Gamma_2| \Vdash y_{\beta}:A_2)$, we have $(\Gamma_1 \Vdash \mathcal{D}_1 :: x_{\alpha}:A_1[c_1]) \equiv_{\xi}^{\Psi_0} (\Gamma_2 \Vdash \mathcal{D}_2 :: y_{\beta}:A_2[c_2])$ iff $(\Gamma_1 \Vdash \mathcal{D}_1 :: x_{\alpha}:A_1[c_1]) \approx_a^{\xi} (\Gamma_2 \Vdash \mathcal{D}_2 :: y_{\beta}:A_2[c_2])$.

Proof. The full proof is available in [5].

Theorem 2 ($\top\top$ -closure). Consider $\mathcal{D}_1 \in \text{Tree}(|\Gamma_1| \Vdash |x_{\alpha}:A_1[c_1]|)$ and $\mathcal{D}_2 \in \text{Tree}(|\Gamma_2| \Vdash |y_{\beta}:A_2[c_2]|)$ and a given observer level $\xi \in \Psi_0$. We have

$$\begin{aligned} (\Gamma_1 \Vdash \mathcal{D}_1 :: x_{\alpha}:A_1[c_1]) \equiv_{\Psi_0}^{\xi} (\Gamma_2 \Vdash \mathcal{D}_2 :: y_{\beta}:A_2[c_2]) \text{ iff} \\ \forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{F}_1, \mathcal{F}_2. \\ \text{if } (\Gamma_1 \dashv \mathcal{C}_1[] \mathcal{F}_1 \dashv x_{\alpha}:A_1[c_1]) \equiv_{\Psi_0}^{\xi} (\Gamma_2 \dashv \mathcal{C}_2[] \mathcal{F}_2 \dashv y_{\beta}:A_2[c_2]) \\ \text{then } \forall \mathcal{B}_1 \in \text{H-Provider}^{\xi}(\Gamma_1). \forall \mathcal{B}_2 \in \text{H-Provider}^{\xi}(\Gamma_2). \\ \forall \mathcal{T}_1 \in \text{H-Client}^{\xi}(x_{\alpha}:A_1[c_1]). \forall \mathcal{T}_2 \in \text{H-Client}^{\xi}(y_{\beta}:A_2[c_2]). \\ \mathcal{B}_1 \mathcal{C}_1 \mathcal{D}_1 \mathcal{F}_1 \mathcal{T}_1 \approx_a \mathcal{B}_2 \mathcal{C}_2 \mathcal{D}_2 \mathcal{F}_2 \mathcal{T}_2 \end{aligned}$$

Proof. The full proof is available in [5].

Completeness ensures that the logical relation relates all secure programs, while IFC type systems, by nature, reject infinitely many secure programs. Our logical relation thus becomes generally applicable. For example, we may conceive a more permissive IFC refinement type system that accepts more secure programs. All we would have to do in this case is prove the fundamental theorem for that new refinement type system, guaranteeing that well-typed programs are self-related by our logical relation and thus non-interfering. We conclude this section by noting that all the results presented here are corollaries of the results in Section 3.

5 Related work and discussion

Logical relations for session types. The application of logical relations to session types has focused predominantly on unary logical relations for proving termination [19, 49, 50, 54], except for a binary logical relation for parametricity [11] and noninterference [17, 18, 27]. Our RSLR shares its foundation in linear logic with this line of work, but contributes significantly through its support of general recursive types.

Caires et al. [11] show parametricity for a terminating language and restrict observations to the single offering channel of a configuration, i.e., observations are only made at the root. Thanks to the validity of semantic cut, a single-type-indexed logical relation, such as the authors', falls out as a special case of our RSLR. However, more fine-grained program equivalences, such as noninterference, demand distinguishing the two roles a process configuration may assume. To the best of our understanding, our account of duality ultimately also enabled biorthogonal closure to prove the logical equivalence sound and complete.

Among prior work on logical relations for ILLST, the work by Derakhshan et al. [17] is most closely related to ours. The authors contribute an IFC session type system and develop a logical relation to show noninterference. The refinement type system that we use in our case study (§ 4.3) is based on that IFC system, extended with secrecy-polymorphic processes [18]. Besides the authors' confinement to a terminating language, sidestepping the intricacies of non-termination and nondeterminism, the authors' metatheoretic results are significantly more limited than ours. In particular, the logical relation is dependent on IFC typing, and thus only amounts to a *syntactic* logical relation for noninterference. Moreover, their development lacks metatheoretic results about the entailed program equivalence, i.e., that it is sound with respect to bisimilarity. There are several nuances in the design of our logical relation that enable us to show such results in a more general setting, which also allows for non-termination: Unlike [17], which only considers closed configurations, i.e., it cannot relate two programs without their clients and providers, our logical relation is defined over open configurations and thus allows for connection to biorthogonality. Furthermore, our term interpretation carefully uses the universal and existential quantifiers and thus enables the proof of transitivity for the logical relation, which is not proven in [17]. Our logical relation also handles non-termination via focus channels and the observation index. These features enable defining the logical relation for equivalence in a general context, proving that it is both sound and complete with respect to asynchronous bisimilarity by establishing a connection to biorthogonality, and then instantiating it in various settings, such as noninterference.

Also noteworthy is the work by Rocha and Caires [53, 54]. The authors contribute CLASS, a calculus based on classical linear logic session types enriched with memory cells, acting as mutexes to share affine data between processes. Unlike more liberal support of sharing [6–8], the authors adopt the propositions-as-types paradigm to ensure deadlock freedom and strong normalization. These guarantees are realized by stratifying resources into nested hierarchies. Interesting in relation to our work, is CLASS' metatheory: to prove strong normalization, the authors develop a unary logical relation, defined to be $\top\top$ -closed reliant on Girard's orthogonality [24]. Besides being binary to express program equivalence and admitting open configurations, our logical relation is not defined to be $\top\top$ -closed, but the logical equivalence induced by our relation shown to be $\top\top$ -closed (Thm. 1). As discussed in § 3.5, the proof of admissibility of $\top\top$ closure serves a validation of the equivalence, guaranteeing that its discriminatory power is strong enough (soundness) while being maximally permissive (completeness).

Relationship to logical relations for stateful languages. Logical relations have been scaled to accommodate state using Kripke logical relations (KLRs) [52]. KLRs are indexed by a *possible world* W , providing a semantic model of the heap. Invariants can then be imposed that must be preserved by any future worlds W' . KLRs can be combined with step indexing [2, 4] to address circularity arising from higher-order stores [3, 21, 22] and to express state transition systems. For example, Gregersen et al. [25] use the KLR supported in Iris [35] to prove noninterference for a functional language with a higher-order store, recursive types, and impredicative polymorphism. The authors also adopt semantic typing [15, 42, 62], admitting syntactically ill-typed programs, if shown to inhabit the logical relation. Besides the difference in language, the authors consider termination-*insensitive* noninterference, whereas we consider progress-*sensitive* noninterference. Like KLRs, our RSLR is situated in a stateful setting because channels, like locations, are subject to concurrent mutation. However, our RSLR is rooted in ILLST, which guarantees race freedom, stratifies the store (the configuration of processes), and prescribes state transitions. When squinting one's eyes, the sequent $\Delta \Vdash K$ of the logical relation seems reminiscent of a possible world, since it provides the semantic typing of the free channels. However, linearity would not comport well with the usual monotonicity requirement of logical relations. We would like to investigate this connection as future work.

6 Concluding remarks

We have contributed a recursive session logical relation (RSLR) for progress-sensitive equivalence of programs. The RSLR is rooted in intuitionistic linear logic session types, inheriting from it a strong foundation in linear logic, ensuring that channel endpoints are treated as resources. A novel aspect of the RSLR is its use of an *observation index* to keep the logical relation well-defined in the presence of general recursive types. This shift, from a step index / unfolding index associated with *internal* computation steps to an index only associated with *external* observations, facilitates statement and proof of metatheoretic properties of the logical relation: closure under parallel composition, soundness and completeness with regard to weak asynchronous bisimilarity, and biorthogonal closure of the logical equivalence relation induced by the RSLR. A biorthogonality argument arises naturally from the duality inherent to session types. In future work, we would like to explore these connections more deeply, possibly connecting to Kripke logical relations and game semantics.

Acknowledgments. This material is based upon work supported by the Air Force Office of Scientific Research under Award No. FA9550-21-1-0385 (Tristan Nguyen, program manager) and upon work supported by the National Science Foundation under Grant No. 2442461. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Department of Defense or the National Science Foundation.

References

1. Ahmed, A.: Semantics of Types for Mutable State. Ph.D. thesis, Princeton University (2004)
2. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: 15th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 3924, pp. 69–83. Springer (2006). https://doi.org/10.1007/11693024_6
3. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 340–353. ACM (2009). <https://doi.org/10.1145/1480881.1480925>
4. Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **23**(5), 657–683 (2001). <https://doi.org/10.1145/504709.504712>
5. Balzer, S., Derakhshan, F., Harper, R., Yao, Y.: Logical relations for session-typed concurrency. *CoRR abs/2309.00192* (2026). <https://doi.org/10.48550/ARXIV.2309.00192>
6. Balzer, S., Pfenning, F.: Manifest sharing with session types. *Proceedings of the ACM on Programming Languages* **1**(ICFP), 37:1–37:29 (2017). <https://doi.org/10.1145/3110281>
7. Balzer, S., Pfenning, F., Toninho, B.: A universal session type for untyped asynchronous communication. In: 29th International Conference on Concurrency Theory (CONCUR). *LIPICs*, vol. 118, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). <https://doi.org/10.4230/LIPICS.CONCUR.2018.30>
8. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: 28th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 11423, pp. 611–639. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_22
9. Benton, N., Hur, C.: Biorthogonality, step-indexing and compiler correctness. In: 14th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 97–108. ACM (2009). <https://doi.org/10.1145/1596550.1596567>
10. Benton, N., Hur, C.: Step-indexing: The good, the bad and the ugly. In: Modelling, Controlling and Reasoning About State. Dagstuhl Seminar Proceedings, vol. 10351, pp. 1–9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2010), <http://drops.dagstuhl.de/opus/volltexte/2010/2808/>
11. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: 22nd European Symposium on Programming (ESOP). pp. 330–349 (2013). https://doi.org/10.1007/978-3-642-37036-6_19
12. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: 21th International Conference on Concurrency Theory (CONCUR). Lecture Notes in Computer Science, vol. 6269, pp. 222–236. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_16
13. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. *Mathematical Structures in Computer Science* **26**(3), 367–423 (2016). <https://doi.org/10.1017/S0960129514000218>

14. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. *Information and Computation* **207**(10), 1044–1077 (2009). <https://doi.org/10.1016/j.ic.2008.11.006>
15. Constable, R.L., Allen, S.F., Bromley, M., Cleaveland, R., Cremer, J.F., Harper, R., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall (1986), <http://dl.acm.org/citation.cfm?id=10510>
16. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. pp. 50–63. ACM (1999). <https://doi.org/10.1145/301618.301641>
17. Derakhshan, F., Balzer, S., Jia, L.: Session logical relations for noninterference. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. pp. 1–14. IEEE Computer Society (2021). <https://doi.org/10.1109/LICS52264.2021.9470654>
18. Derakhshan, F., Balzer, S., Yao, Y.: Regrading policies for flexible information flow control in session-typed concurrency. In: *38th European Conference on Object-Oriented Programming (ECOOP)*. *LIPICs*, vol. 313, pp. 11:1–11:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). <https://doi.org/10.4230/LIPICs.ECOOP.2024.11>
19. DeYoung, H., Pfenning, F., Pruiksmas, K.: Semi-axiomatic sequent calculus. In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD)*. *LIPICs*, vol. 167, pp. 29:1–29:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPICs.FSCD.2020.29>
20. Dreyer, D., Ahmed, A., Birkedal, L.: Logical step-indexed logical relations. In: *24th Annual IEEE Symposium on Logic in Computer Science (LICS)*. pp. 71–80. IEEE Computer Society (2009). <https://doi.org/10.1109/LICS.2009.34>
21. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. In: *15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. pp. 143–156. ACM (2010). <https://doi.org/10.1145/1863543.1863566>
22. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* **22**(4-5), 477–528 (2012). <https://doi.org/10.1017/S095679681200024X>
23. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. *Acta Informatica* **42**(2-3), 191–225 (2005). <https://doi.org/10.1007/s00236-005-0177-z>
24. Girard, J.: Linear logic. *Theoretical Computer Science* **50**, 1–102 (1987). [https://doi.org/10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4)
25. Gregersen, S.O., Bay, J., Timany, A., Birkedal, L.: Mechanized logical relations for termination-insensitive noninterference. *Proceedings of the ACM on Programming Languages* **5**(POPL), 1–29 (2021). <https://doi.org/10.1145/3434291>
26. Heidin, D., Sabelfeld, A.: *A perspective on information-flow control*. Tech. rep., Marktoberdorf (2011)
27. van den Heuvel, B., Derakhshan, F., Balzer, S.: Information flow control in cyclic process networks. In: *38th European Conference on Object-Oriented Programming (ECOOP)*. *LIPICs*, vol. 313, pp. 40:1–40:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). <https://doi.org/10.4230/LIPICs.ECOOP.2024.40>
28. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall (1985)
29. Honda, K.: Types for dyadic interaction. In: *4th International Conference on Concurrency Theory (CONCUR)*. *Lecture Notes in Computer Science*, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35

30. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: 7th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). <https://doi.org/10.1007/BFb0053567>
31. Hur, C., Dreyer, D.: A kripke logical relation between ML and assembly. In: 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 133–146. ACM (2011). <https://doi.org/10.1145/1926385.1926402>
32. Hur, C., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and kripke logical relations. In: 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 59–72. ACM (2012). <https://doi.org/10.1145/2103656.2103666>
33. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. In: 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 128–141 (2001). <https://doi.org/10.1145/360204.360215>
34. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. *Theoretical Computer Science* **311**(1-3), 121–163 (2004). [https://doi.org/10.1016/S0304-3975\(03\)00325-6](https://doi.org/10.1016/S0304-3975(03)00325-6)
35. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* **28**, e20 (2018). <https://doi.org/10.1017/S0956796818000151>
36. Kobayashi, N.: A partially deadlock-free typed process calculus. In: 12th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 128–139. IEEE Computer Society (1997). <https://doi.org/10.1109/LICS.1997.614941>
37. Kobayashi, N.: A new type system for deadlock-free processes. In: 17th International Conference on Concurrency Theory (CONCUR). LNCS, vol. 4137, pp. 233–247 (2006). https://doi.org/10.1007/11817949_16
38. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: A fully-abstract semantics for classical processes. *Proceedings of the ACM on Programming Languages* **3**(POPL), 24:1–24:29 (2019). <https://doi.org/10.1145/3290337>
39. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 141–152. ACM (2006). <https://doi.org/10.1145/1111037.1111050>
40. Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: 24th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 9032, pp. 560–584. Springer (2015). https://doi.org/10.1007/978-3-662-46669-8_23
41. Lindley, S., Morris, J.G.: Talking bananas: Structural recursion for session types. In: 21st ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 434–447. ACM (2016). <https://doi.org/10.1145/2951913.2951921>
42. Martin-Löf, P.: Constructive mathematics and computer programming. In: *Logic, Methodology and Philosophy of Science VI, Studies in Logic and the Foundations of Mathematics*, vol. 104, pp. 153–175. Elsevier (1982). [https://doi.org/https://doi.org/10.1016/S0049-237X\(09\)70189-2](https://doi.org/10.1016/S0049-237X(09)70189-2)
43. Melliès, P., Vouillon, J.: Recursive polymorphic types and parametricity in an operational framework. In: 20th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 82–91. IEEE Computer Society (2005). <https://doi.org/10.1109/LICS.2005.42>
44. Milner, R.: *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92. Springer (1980). <https://doi.org/10.1007/3-540-10235-3>

45. Milner, R.: Communication and Concurrency. PHI Series in Computer Science, Prentice Hall (1989)
46. Milner, R.: Communicating and Mobile Systems: the π -calculus. Cambridge University Press (1999)
47. Morris, J.H.: Lambda-Calculus Models of Programming Languages. Ph.D. thesis, Massachusetts Institute of Technology (1968)
48. Neis, G., Dreyer, D., Rossberg, A.: Non-parametric parametricity. *Journal of Functional Programming* **21**(4-5), 497–562 (2011). <https://doi.org/10.1017/S0956796811000165>
49. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: 21st European Symposium on Programming (ESOP). *Lecture Notes in Computer Science*, vol. 7211, pp. 539–558. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_27
50. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation* **239**, 254–302 (2014). <https://doi.org/10.1016/j.ic.2014.08.001>
51. Pitts, A.M.: Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* **10**(3), 321–359 (2000), <http://journals.cambridge.org/action/displayAbstract?aid=44651>
52. Pitts, A.M., Stark, I.: Operational reasoning for functions with local state. *Higher Order Operational Techniques in Semantics (HOOTS)* pp. 227–273 (1998)
53. Rocha, P.: CLASS: A Logical Foundation for Typeful Programming with Shared State. Ph.D. thesis, NOVA University Lisbon (2022)
54. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state. In: 32nd European Symposium on Programming (ESOP). *Lecture Notes in Computer Science*, vol. 13990, pp. 421–450. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_16
55. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE Journal of Selected Areas in Communications* **21**(1), 5–19 (2003). <https://doi.org/10.1109/JSAC.2002.806121>
56. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: 22nd Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 293–302. IEEE Computer Society (2007). <https://doi.org/10.1109/LICS.2007.17>
57. Sangiorgi, D., Walker, D.: The π -calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
58. Smith, G., Volpano, D.M.: Secure information flow in a multi-threaded imperative language. In: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 355–364. ACM (1998). <https://doi.org/10.1145/268946.268975>
59. Støvring, K., Lassen, S.B.: A complete, co-inductive syntactic theory of sequential control and state. In: 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 161–172. ACM (2007). <https://doi.org/10.1145/1190216.1190244>
60. Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. *Journal of the ACM* **54**(5), 26 (2007). <https://doi.org/10.1145/1284320.1284325>
61. Thamsborg, J., Birkedal, L.: A kripke logical relation for effect-based program transformations. In: 16th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 445–456. ACM (2011). <https://doi.org/10.1145/2034773.2034831>

62. Timany, A., Krebbers, R., Dreyer, D., Birkedal, L.: A logical approach to type soundness. *Journal of the ACM (JACM)* **71**(6), 40:1–40:75 (2024). <https://doi.org/10.1145/3676954>
63. Toninho, B.: A Logical Foundation for Session-Based Concurrent Computation. Ph.D. thesis, Carnegie Mellon University and New University of Lisbon (2015)
64. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: 22nd European Symposium on Programming (ESOP). *Lecture Notes in Computer Science*, vol. 7792, pp. 350–369. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_20
65. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. *Journal of Computer Security* **4**(2/3), 167–188 (1996). <https://doi.org/10.3233/JCS-1996-42-304>
66. Wadler, P.: Propositions as sessions. In: ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 273–286. ACM (2012). <https://doi.org/10.1145/2364527.2364568>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

