Declarative Interpretations of Session-Based Concurrency

Mauricio Cano and Camilo Rueda
Pontificia Universidad Javeriana - Cali

Hugo A. López
LaSIGE, University of Lisbon

Jorge A. Pérez
University of Groningen

Abstract

Session-based concurrency is a type-based approach to the analysis of communication-intensive systems. Correct behavior in these systems may be specified in an *operational* or *declarative* style: the former defines how interactions are structured; the latter defines governing conditions. In this paper, we investigate the relationship between operational and declarative models of session-based concurrency. We propose two interpretations of session π-calculus processes as declarative processes in linear concurrent constraint programming (`lcc`). They offer a basis on which both operational and declarative requirements can be specified and reasoned about. By coupling our interpretations with a type system for `lcc`, we obtain robust declarative encodings of π-calculus mobility.

Categories and Subject Descriptors D.3.1 [*Programming languages*]: Formal Definitions and Theory; F.3.2 [*Logics and meanings of programs*]: Semantics of Programming Languages–Process models

Keywords concurrency, π-calculus, concurrent constraint programming, session types, eabbrvnatxpressiveness.

1. Introduction

This paper relates two distinct models of concurrent processes: one of them, the session π-calculus (sπ [27]), is inherently *operational*; the other one, given by linear concurrent constraint programming (`lcc` [9, 13]), is *declarative*. Our interest is in the analysis of *communication-intensive* systems, which are best described by combining features from both paradigms. We aim at formal relationships in terms of expressiveness, as these are the basis for the sound transference of reasoning and validation techniques. In this work, the common trait supporting such relationships is *linearity*.

Session-based concurrency is a type-based approach to the analysis of communication-intensive systems. Structured dialogues (*protocols*) are organized into basic units called *sessions*; interaction patterns are abstracted as *session types* [16], against which specifications may be checked. As these specifications can be conveniently given in the π-calculus [21], we obtain collections of processes interacting along names/channels. A session connects exactly two partners and is characterized by two distinct phases. In the first one, processes requesting/offering protocols seek a complementary (dual) partner; the second phase occurs as soon as two partners agree to interact according to some session protocol. Sessions combine concurrency, mobility, and resource-awareness: while the first phase may be non-deterministic and uses unrestricted *(service) names*, the second one is characterized by deterministic interaction sequences along linear *(session) channels*.

In the realm of structured communications, *operational* and *declarative* approaches are complementary: while operational approaches describe *how* a communicating system is implemented, declarative ones describe *what* are the (minimum) conditions that govern correct behavior. Although the operational concurrency of the π-calculus is convenient to specify directed, mobile communications, expressing other kinds of requirements that influence interaction, in particular partial/contextual information on structured protocols and partners, is unnatural or too convoluted. This is not a new observation: several previous works have proposed declarative extensions of name-passing calculi (e.g., [3, 6, 8]). On the other hand, declarative models of concurrency naturally express partial and contextual information (see, e.g., [7, 23, 25]). Although some of these models may represent forms of π-calculus mobility [18, 25], such representations may be unpractical to work with.

In our view, all of the above begs for a *unifying* account of operational and declarative approaches to session-based concurrency, so as to articulate existing languages and analysis techniques at appropriate abstraction levels. This includes formally relating different languages, which in turn should enable the *sound transference* of verification techniques across operational and declarative models.

In a previous work [19] we described a first step towards such a unified account of operational and declarative approaches. We gave an encoding of the session π-calculus in [16] as declarative processes in *universal* concurrent constraint programming (`utcc`) [25]. While insightful, this encoding has two limitations:

(a) The key role of *linearity* in session-based concurrency is not explicit in declarative encodings of session π-calculus processes.

(b) Declarative encodings of mobility and scope extrusion in `utcc`, based on the *abstraction* operator, are not robust enough to properly match their operational counterparts in the π-calculus.

In this paper, we propose two interpretations of sπ processes as declarative processes in *linear* concurrent constraint programming (`lcc` [9, 13]). Our interpretations follow the approach in [19], but enhance it significantly by addressing the above limitations. As for (a), an immediate consequence of moving to `lcc` is that the encodings given here offer a clean treatment of linearity as essential in operational processes. This in turn leads to more precise operational correspondence results connecting both paradigms. Moreover, the connection with `lcc` enables alternative approaches to behavioral equivalences [13] and to the verification of safety properties for session-based processes [9, 14]. As for (b), we address this anomaly by considering a *type system* for `lcc` processes. Building upon [15], our type system distinguishes the variables available for (linear) abstractions. By stipulating precisely which variables can be abstracted and which cannot, we can limit the power of abstraction so to faithfully represent hiding and scope extrusion in $\mathtt{s}\pi$.

Next, we illustrate further our approach and contributions. § 3 introduces $\mathtt{s}\pi$ and lcc. A first interpretation of $\mathtt{s}\pi$ in lcc is given in § 4. The type system for lcc is presented in § 5. Our second interpretation, given in § 6, considers $\mathtt{s}\pi^{+}$: the extension of $\mathtt{s}\pi$ with session establishment. On top of this interpretation, we use types for lcc to ensure correct abstractions. We close by discussing related work (§ 7) and collecting some concluding remarks (§ 8).

2. Overview

A Basic Scenario. Consider a structured interaction between a client and an online store, as given by the following description:

1. The client sends a description of an item to the store.
2. The store replies with the price of the item and offers two options to the client: to buy the item or to close the transaction.
3. Depending on the price of the item, the client purchases the item or ends the transaction.

This simple structured interaction may be expressed using session types as follows. From the client's perspective, the session type

$$T_c = !\mathsf{item}.\ ?\mathsf{price}.\ \oplus\{\text{buy} :\ B_c\ ,\ \text{close} : !\mathsf{bye}.\,end\}$$

says that the output (!) of a value of type item, should be followed by the input (?) of a value of type price. These exchanges precede the selection ($\oplus$) between two behaviors denoted by *labels* buy and close, and abstracted by types B_c and $!\mathsf{bye}.\,end$, respectively.

We illustrate the relationship between session types and π-calculus processes; we first introduce some notation. We write $\overline{x}\,v.P$ and $x(y).P$ to denote output and input along name x, with continuation P. Also, we write $x \triangleleft \text{lab}.\,P$ to represent selection of a label lab, and $b?\,(P)\,\mathbf{:}\,(Q)$ to denote a conditional expression (which executes P or Q depending on boolean b). The following process could be an implementation of T_c above, along x:

$$P_x = \overline{x}\,\mathsf{book}.\,x(z).\,(z \leq 20)?\,(x{\triangleleft}\text{buy}.\,R_c)\,\mathbf{:}\,(x{\triangleleft}\text{close}.\,\overline{x}\,\text{end}.\,\mathbf{0})$$

where R_c is the client implementation of the purchase routine.

The relationship between session types (such as T_c) and process implementations in the π-calculus (such as P_x) has been thoroughly studied; the way in which session type checking can enforce non-trivial communication properties on processes (e.g., protocol compliance, deadlock-freedom) is rather well-understood by now. In particular, the key role that *linearity* (and linear logic at large) plays in session type systems has been recently clarified [4, 28].

The Basic Scenario with Declarative Conditions. The session π-calculus appropriately describes the communication layer of protocols, but may be less adequate to specify conditions that influence partners and interactions. As an example, consider a protocol that is as the one given above, except for its last step, which is as follows:

3'. Depending on *both* the item's price *and on the occurrence of event* e, the client purchases the item or ends the transaction.

Events (and event detection) are not unusual features in structured protocols; their type-based analysis has been studied in [17]. In the modified protocol, event e may represent a contextual condition on the system's state, say a flag triggered when a variable falls within some threshold (an indicator of *partial information*). Independently of the actual event, it is clear that although event e influence communication behavior (i.e., in deciding to purchase the item or not) it can be hardly considered as a communication action. This is why *declarative requirements* (of which event detection is just but an instance) appear as unnatural features to express in the π-calculus. We thus argue that the (standard) π-calculus does not naturally lend itself to specify the combination of operational descriptions of structured interactions (typical of sessions) and declarative requirements (typical of, e.g., protocol and workflow specifications).

Our Approach. We are thus interested in formalisms in which operational and declarative requirements can be jointly specified. We focus on process models based on *concurrent constraint programming* (ccp) [26]. In ccp, processes interact via a *global store* by means of *tell* and *ask* operations. Processes may add new constraints (pieces of partial information) to the store by means of tell operations; using ask operations processes may also query the store about some constraint and react accordingly.

Here we study how a particular process model based on ccp can provide a unified basis for specifying and reasoning about session-based concurrency. The languages that we consider are $\mathtt{s}\pi$, the session π-calculus in [27] (§ 3.1), and lcc [9, 13] (§ 3.2). We introduce two declarative interpretations (encodings) of $\mathtt{s}\pi$ into lcc and establish their properties. Although establishing correctness of these interpretations is insightful in itself, an important related issue is understanding to what extent the properties of $\mathtt{s}\pi$ can be transposed to the declarative world of lcc through our interpretations.

Our Contributions. A common trait in $\mathtt{s}\pi$ and lcc is *linearity*: in session-based concurrency, it enables the enforcement of disciplined resource-aware protocols (via session types); linearity is also central to lcc, as we explain next.

Let c, d and $\vec{x}$ denote constraints and a (possibly empty) vector of variables. In lcc, constructs for tell and ask operations are denoted as $\overline{c}$ and $\forall\vec{x}(d \to P)$, respectively. (When $\vec{x}$ is empty, we write $\forall\epsilon(d \to P)$.) While process $\overline{c}$ can be seen as the output of c to the store, process $\forall\vec{x}(d \to P)$ (a *linear abstraction*) may be intuitively read as: if constraint d can be inferred from the current store then P will be executed. This inference consumes the abstraction; it may also involve consumption of constraints in the store and substitution of $\vec{x}$ in P, cf. § 3.2.

In this work, we develop two interpretations (*encodings*) of $\mathtt{s}\pi$ into lcc. In the *first interpretation* (denoted $[\![\cdot]\!]$ and given in § 4), output $\overline{x}\,v.P$ and input $x(y).Q$ in $\mathtt{s}\pi$ are encoded as

$$[\![\overline{x}\,v.P]\!] = \overline{out(x,v)} \parallel \forall z\big((in(z,v) \otimes \{x{:}z\}) \to [\![P]\!]\big)$$
$$[\![x(y).Q]\!] = \forall y, w\big((out(w,y) \otimes \{w{:}x\}) \to \overline{in(x,y)} \parallel [\![Q]\!]\big)$$

Predicates $out(x,v)$ and $in(x,y)$ are used to model synchronous communication in $\mathtt{s}\pi$; constraint $\{x{:}z\}$ says that x and z are two dual *session endpoints*. These pieces of information are treated as linear resources by lcc; this is critical to ensure faithfulness of the interpretation with respect to the source $\mathtt{s}\pi$ process (cf. Thm. 4.12). This interpretation attests the expressivity of linear abstractions in representing name passing and scope extrusion in $\mathtt{s}\pi$.

Using $[\![\cdot]\!]$ we can already give lcc specifications which combine representations of $\mathtt{s}\pi$ communication and declarative requirements, using partial information based on constraints. We may, e.g., "plug" such representations into declarative contexts that specify behaviors hard to specify in $\mathtt{s}\pi$. As a simple example, consider a fragment of P_x above. Let $B = x \triangleleft \text{buy}.\,R_c$ and $A = x \triangleleft \text{close}.\,\overline{x}\,\text{end}.\,\mathbf{0}$. In our approach, we could have, e.g., the lcc process

$$\forall\epsilon(e \wedge z > 20 \to [\![A]\!]) \parallel \forall\epsilon(e \wedge z \leq 20 \to [\![B]\!])$$

which uses conjunction to add the presence of event e into the decision of abandoning the protocol ($[\![A]\!]$) or buying the item ($[\![B]\!]$).

It turns out that linear abstractions are overly powerful: they may express forms of scope extrusion not possible in $\mathtt{s}\pi$ (see § 5.2). To overcome this anomaly, our *second interpretation*, given in § 6, encodes an extension of $\mathtt{s}\pi$ with session establishment ($\mathtt{s}\pi^{+}$) using *linear abstractions with local information*:

$$\forall\vec{x}(d\,\mathbf{;}\,e \to P)$$

$$
\begin{array}{ll}
\lfloor\textsc{Com}\rfloor & (\boldsymbol{\nu}xy)(\overline{x}\,v.P \mid y(z).Q) \to_\pi (\boldsymbol{\nu}xy)(P \mid Q\{^v\!/_z\})\\
\lfloor\textsc{Rep}\rfloor & (\boldsymbol{\nu}xy)(\overline{x}\,v.P \mid *\,y(z).Q) \to_\pi (\boldsymbol{\nu}xy)(P \mid Q\{^v\!/_z\} \mid *\,y(z).Q)\\
\lfloor\textsc{Sel}\rfloor & (\boldsymbol{\nu}xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I}) \to_\pi (\boldsymbol{\nu}xy)(P \mid Q_j)\ (j \in I)\\
\lfloor\textsc{IfT}\rfloor & \texttt{tt}?\,(P)\,\boldsymbol{:}\,(Q) \to_\pi P\\
\lfloor\textsc{IfF}\rfloor & \texttt{ff}?\,(P)\,\boldsymbol{:}\,(Q) \to_\pi Q\\
\lfloor\textsc{Res}\rfloor & P \to_\pi P' \Rightarrow (\boldsymbol{\nu}xy)P \to_\pi (\boldsymbol{\nu}xy)P'\\
\lfloor\textsc{Par}\rfloor & P \to_\pi P' \Rightarrow P \mid R \to_\pi P' \mid R\\
\lfloor\textsc{Str}\rfloor & P \equiv_\pi P',\ P' \to_\pi Q',\ Q' \equiv_\pi Q \Rightarrow P \to_\pi Q
\end{array}
$$

Figure 1. Reduction relation for $\mathsf{s}\pi$ processes.

where d is a piece of local information (e.g., a session key) used jointly with e to trigger P. Abstractions with local information refine the abstractions in [13], which act on the global store. The use of local information in abstractions does not suffice to properly limit their expressivity. We then couple our second interpretation with a *type system* on `lcc` processes (cf. § 5), which controls the use of variables in abstractions. Precisely, we identify *secure patterns* in abstractions; a well-typed `lcc` process is one that contains only abstractions with secure patterns. We show that our second interpretation is well-typed, which rules out (malicious) abstractions that may interfere with encodings of name mobility.

3. Preliminaries

We now introduce the models that we formally relate in this work: the session π-calculus of [27] (§ 3.1) and `lcc` [13] (§ 3.2). Notation $\vec{e}$ denotes a sequence of elements $e_1, \dots, e_n$ with length $|\vec{e}\,| = n$.

3.1 The session π-calculus ($\mathsf{s}\pi$)

Syntax. Assume a countable infinite set of variables $\mathcal{V}_\pi$, ranged over by $x, y, \dots$. For simplicity, we only consider boolean constants ($\texttt{tt}, \texttt{ff}$); we use $v, v', \dots$ to range over variables and constants (*values*). Also, we use $l, l', \dots$ to range over *labels*.

Definition 3.1 ($\mathsf{s}\pi$ Processes). *The syntax for $\mathsf{s}\pi$ processes is given by the following grammar:*

$$
\begin{array}{rcl}
P, Q & ::= & \overline{x}\,v.P \mid x(y).P \mid x \triangleleft l.P \mid x \triangleright \{l_i : P_i\}_{i\in I}\\
 & \mid & *\,x(y).P \mid v?\,(P)\,\boldsymbol{:}\,(Q) \mid P \mid Q \mid (\boldsymbol{\nu}xy)P \mid \mathbf{0}
\end{array}
$$

Process $\overline{x}\,v.P$ sends value v over channel x and then continues as P; dually, the process $x(y).Q$ expects a value v on x that will replace free occurrences of y in Q. Process $x \triangleleft l_j.P$ uses x to select l_j from a labeled choice process $x \triangleright \{l_i : P_i\}_{i\in I}$ so as to trigger P_j. We assume pairwise distinct labels. Process $*\,x(y).P$ denotes a replicated input process; it allows one to express infinite behaviors. The conditional $v?\,(P)\,\boldsymbol{:}\,(Q)$ is standard: if v evaluates to $\texttt{tt}$ then it behaves as P; otherwise it behaves as Q. Constructs for parallel composition and inaction are also standard. The construct for restriction, $(\boldsymbol{\nu}xy)P$, is the main difference with respect to usual π-calculus presentations: it simultaneously binds the *co-variables* (or *session endpoints*) x and y in P. In process $x(y).P$ (resp. $(\boldsymbol{\nu}yz)P$) occurrences of y (resp. y, z) are binding with scope P. The set of *free names* of P is denoted $\mathsf{fn}(P)$.

Operational Semantics. The semantics for $\mathsf{s}\pi$ processes is given as a *reduction relation*, denoted $\to_\pi$, defined as the smallest relation generated by the rules in Fig. 1. Reduction expresses the computation steps that a process performs on its own. It relies on a *structural congruence* on processes, denoted $\equiv_\pi$, which identifies processes up to consistent renaming of bound names, denoted $\equiv_\alpha$.

(*Qualifiers*)	q ::=	lin	(linear)
		un	(unrestricted)
(*Pretypes*)	p ::=	*bool*	(booleans)
		end	(inaction)
		$?T.T$	(receive)
		$!T.T$	(send)
		$\oplus\{l_i : T_i\}_{i\in I}$	(select)
		$\&\{l_i : T_i\}_{i\in I}$	(branching)
(*Types*)	T ::=	$p\,q$	(qualified pretype)
		a	(type variable)
		$\mu a.T$	(recursive type)
(*Contexts*)	Γ ::=	$\emptyset$	(empty context)
		$\Gamma, x : T$	(assumption)

Figure 2. Session types: Qualifiers, Pretypes, Types, Contexts.

Formally, $\equiv_\pi$ is the smallest congruence that satisfies the axioms:

$$
\begin{array}{c}
P \mid \mathbf{0} \equiv_\pi P \qquad P \mid Q \equiv_\pi Q \mid P \qquad P \equiv_\pi Q \text{ if } P \equiv_\alpha Q\\
(P \mid Q) \mid R \equiv_\pi P \mid (Q \mid R) \qquad (\boldsymbol{\nu}xy)(\boldsymbol{\nu}yz)P \equiv_\pi (\boldsymbol{\nu}yz)(\boldsymbol{\nu}xy)P\\
(\boldsymbol{\nu}xy)\mathbf{0} \equiv_\pi \mathbf{0} \qquad (\boldsymbol{\nu}xy)P \mid Q \equiv_\pi (\boldsymbol{\nu}xy)(P \mid Q) \text{ if } x, y \notin \mathsf{fn}(Q)
\end{array}
$$

Type System. We summarize the type system presented in [27]. Still, the paper can be read without knowing its details.

Definition 3.2 (Session Types: Syntax). *The syntax of session types is given in Fig. 2. Notice that q ranges over qualifiers, p ranges over pretypes, T ranges over types, and Γ denotes contexts.*

Pretype *bool* is used for constants and variables. The pretype end denotes the terminated protocol; it types a channel that can no longer be used. Pretype $!T_1.T_2$ denotes output, and types a channel that sends a value of type T_1 and continues according to type T_2. Dually, pretype $?T_1.T_2$ denotes input, and types a channel that receives a value of type T_1 and then proceeds according to type T_2. Pretypes $\oplus\{l_i : T_i\}_{i\in I}$ and $\&\{l_i : T_i\}_{i\in I}$ denote labeled selection (internal choice) and branching (external choice), respectively.

Types are *qualified* pretypes or recursive types for disciplining potentially infinite communication patterns. Intuitively, linearly qualified types are assigned to endpoints occurring in exactly one thread (a process not comprising parallel composition); the unrestricted qualifier allows an endpoint to occur in multiple threads.

Session type systems depend on *type duality* to relate session types with opposite behaviors: e.g., the dual of input is output (and vice versa); branching is the dual of selection (and vice versa). This intuition suffices for the purposes of this paper; see, e.g., [1] for a formal definition. We write $\overline{T}$ to denote the dual of type T.

Given a context Γ and a process P, typing judgments are of the form $\Gamma \vdash P$. Fig. 3 gives selected typing rules; we now give some intuitions (see [27] for full details). Typing uses a *context splitting* operator on contexts, denoted $\circ$, which maintains the linearity invariant for channels. Rule (T:PAR) types parallel composition using context splitting to divide resources among the two sub-processes. Rule (T:RES) types the restriction operator: it performs a duality check on the types of the co-variables. Rule (T:IN) types an input process: it checks whether x has the right type and checks the continuation; it also adds variable y with type T and x with the type of the continuation to the context. Rule (T:OUT) splits the context in three parts: the first is used to check the type of the sent object; the second is used to check the type of subject; the third is used to check the continuation. Rules (T:BRA) and (T:SEL) type-check branching and selection processes, respectively.

We state the *subject reduction* property for this type system:

Theorem 3.3 ([27]). *If $\Gamma \vdash P$ and $P \to_\pi Q$ then $\Gamma \vdash Q$.*

$$\text{(T:PAR)}\ \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{(T:RES)}\ \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P}$$

$$\text{(T:IN)}\ \frac{\Gamma_1 \vdash x : q\,?T.U \quad (\Gamma_2, y : T) \circ x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}$$

$$\text{(T:OUT)}\ \frac{\Gamma_1 \vdash x : q\,!T.U \quad \Gamma_2 \vdash v : T \quad \Gamma_3 \circ x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}\, v.P}$$

$$\text{(T:SEL)}\ \frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \quad \Gamma_2 \circ x : T_j \vdash P \quad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P}$$

$$\text{(T:BRA)}\ \frac{\Gamma_1 \vdash x : q \;\&\; \{l_i : T_i\}_{i \in I} \quad \forall i \in I.\, \Gamma_2 \circ x : T_i \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i \in I}}$$

Figure 3. Session types: Selected typing rules for $\mathtt{s}\pi$ processes.

We now collect some results that concern the structure of processes, following [27]. Some auxiliary notions are needed. We say $\overline{x}\, v.P$, $x(y).P$, $x \triangleleft l.P$, $x \triangleright \{l_i : P_i\}_{i \in I}$, and $*\, x(y).P$ are processes *prefixed at variable x*. *Redexes* are processes of the form $\overline{x}\, v.P \mid y(z).Q$, $\overline{x}\, v.P \mid *\, y(z).Q$, or $x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i \in I}$, with $j \in I$. We then define *well-formed processes*:

Definition 3.4 (Well-formed process). *An $\mathtt{s}\pi$ process P_0 is* well-formed *if for each of its structural congruent processes*

$$P_0 \equiv_\pi (\boldsymbol{\nu} x_1 y_1) \ldots (\boldsymbol{\nu} x_m y_m)(P \mid Q \mid R) \qquad (m \geq 0)$$

the following conditions hold:

1. *If $P \equiv_\pi v?\,(P')\,\mathbf{:}\,(P'')$ then $v = \mathtt{tt}$ or $v = \mathtt{ff}$.*
2. *If P and Q are prefixed at the same variable, then they are of the same nature (input, output, branch and selection).*
3. *If P is prefixed at x_i and Q is prefixed at y_i, $1 \leq i \leq m$, then $P \mid Q$ is a redex.*

To focus on processes with meaningful forms of interaction, we sometimes consider *programs*:

Notation 3.5 ((Typable) Programs). A process P without free variables is called *a program*. Therefore, program P is typable if it is well-typed under the empty environment ($\vdash P$).

The following result connects programs and well-formedness:

Lemma 3.6 ([27]). *If $\vdash P$ then P is well-formed.*

3.2 Linear Concurrent Constraint Programming (`lcc`)

The linear concurrent constraint calculus (`lcc`) [9, 13] is a declarative formalism based on the `ccp` model that enables reasoning about concurrent systems with partial information and linear resources. As in `ccp` [26], concurrent processes in `lcc` interact via a *global store* by means of *tell* and *ask* operations. The store thus defines a synchronization mechanism. `lcc` has strong ties to linear logic [10] as well as reasoning techniques over processes based on observational equivalences [13] and phase semantics [9]. With respect to other `ccp` languages, there is a key difference: `lcc` allows us to have non-monotonic evolutions of the store, as the ask operator may consume constraints, which are treated as linear resources.

Syntax. We assume countably infinite sets $\mathcal{V}_l$, Σ_c, and Σ_f of variables, predicate symbols, and of functions and constants Σ_f. An arbitrary predicate is denoted γ. First-order terms are built from $\mathcal{V}_l$ and Σ_f will be denoted by t.

Definition 3.7 (`lcc` syntax). *The syntax for `lcc` is given by the following grammar:*

$$\begin{array}{rcl} c & := & \mathtt{1} \mid \mathtt{0} \mid \gamma(\vec{t}) \mid c \otimes c \mid \exists \vec{x}.c \mid\ !c \\ G & := & \forall \vec{x}(c \rightarrow P) \mid G + G \\ P & := & \overline{c} \quad \mid P \parallel Q \mid \exists \vec{x}.\, P \mid\ !\, P \mid G \end{array}$$

The grammar for *constraints* c defines the pieces of information that will be posted (asked) to (from) the store. Constant $\mathtt{1}$, the multiplicative identity, denotes truth; constant $\mathtt{0}$ denotes falsehood. Predicates are denoted $\gamma(\vec{t})$. Formulas are built from the multiplicative conjunction ($\otimes$), bang (!), and the existential quantifier ($\exists \vec{x}$).

Our syntax for *guards* G includes the parametric ask operator $\forall \vec{x}(c \rightarrow P)$ and non-deterministic choice over guards $G_1 + G_2$. When $\vec{x}$ is empty, $\forall \vec{x}(c \rightarrow P)$ is denoted as $\forall \epsilon(c \rightarrow P)$. Processes P include guards as well as the tell operator (denoted $\overline{c}$) and constructs for parallel composition ($\parallel$), hiding ($\exists$), and replication (!), which have expected readings as forms of concurrency, local and infinite behavior, respectively. Notation $\prod_{1 \leq i \leq n} P_i$ (with $n \geq 1$) stands for the process $P_1 \parallel \cdots \parallel P_n$.

Existential and universal quantifiers are variable binders. The free variables of constraints and processes are denoted $fv(\cdot)$. We write $c\{\vec{t}/\vec{x}\}$ to denote the constraint obtained by the (capture-avoiding) substitution of free occurrences of x_i for t_i in c, assuming $|\vec{t}| = |\vec{x}|$ and pairwise distinct x_i's. The process substitution $P\{\vec{t}/\vec{x}\}$ is defined analogously.

Semantics. We follow the semantics for `lcc` processes given by Haemmerlé [13], which is defined as a Labeled Transition System (LTS) and considers a set of linear constraints $\mathcal{C}$ and an entailment relation $\Vdash_C$ over $\mathcal{C}$. We notice that $\mathcal{C}$ is parametric on a given set of predicates, and so $\mathcal{C}$ may change according to signature Σ_c. The LTS relies on a structural congruence on processes, given next.

Definition 3.8 (Structural Congruence). *The structural congruence relation for `lcc` processes is the smallest congruence relation $\equiv_l$ which satisfies the following rules:*

$$P \parallel \overline{\mathtt{1}} \equiv_l P \quad \exists z.\, \overline{\mathtt{1}} \equiv_l \overline{\mathtt{1}} \quad \exists x.\, \exists y.\, P \equiv_l \exists y.\, \exists x.\, P \quad !P \equiv_l P \parallel !P$$

$$\frac{c \otimes d \dashv\Vdash_C e}{\overline{c} \parallel \overline{d} \equiv_l \overline{e}} \qquad \frac{P \equiv_l P'}{P \parallel Q \equiv_l P' \parallel Q}$$

$$\frac{z \notin fv(P)}{P \parallel \exists z.\, Q \equiv_l \exists z.\, (P \parallel Q)} \qquad \frac{P \equiv_l P'}{\exists x.\, P \equiv_l \exists x.\, P'}$$

A transition $P \xrightarrow{\alpha} P'$ denotes the evolution of process P to P' by performing the action denoted by label α:

$$\alpha \quad := \quad \tau \mid c \mid (\vec{x})\overline{c}$$

Label τ denotes a silent (internal) action. Label c denotes a constraint $c \in \mathcal{C}$ "received" as an input action (but see below) and $(\vec{x})\overline{c}$ denotes an output (tell) action in which $\vec{x}$ are extruded variables and $c \in \mathcal{C}$. We write $ev(\alpha)$ to refer to these extruded variables.

The LTS for `lcc` processes is generated by the rules in Fig. 4. The premise $\mathbf{mgc}\big(c, \exists x(d \otimes e)\big)$ in rules (C:OUT) and (C:SYNC) denotes the *most general choice* ($\mathbf{mgc}$) predicate:

Definition 3.9 (Most General Choice (mgc) [13]). *Let c, d, e be constraints, $\vec{x}, \vec{y}$ be vectors of variables and $\vec{t}$ be a vector of terms. We write*

$$\mathbf{mgc}\big(c, \exists \vec{y}(d\{\vec{t}/\vec{x}\} \otimes e)\big)$$

whenever for any constraint e', all terms $\vec{t'}$ and all variables $\vec{y'}$, if $c \Vdash_C \exists \vec{y'}(d\{\vec{t'}/\vec{x}\} \otimes e')$ and $\exists \vec{y'} e' \Vdash_C \exists \vec{y} e$ hold, then $\exists \vec{y}(d\{\vec{t}/\vec{x}\}) \Vdash_C \exists \vec{y'}(d\{\vec{t'}/\vec{x}\})$ and $\exists \vec{y} e \Vdash_C \exists \vec{y'} e'$.

$$\text{(C:OUT)}\quad \frac{c \Vdash_C \exists\vec{x}(d \otimes e) \quad \exists\vec{x}d \Vdash_C \exists\vec{x'}d' \qquad \mathbf{mgc}\big(c, \exists\vec{x}(d \otimes e)\big) \quad (\vec{x} \cup \vec{x'}) \cap fv(c) = \emptyset}{\overline{c} \xrightarrow{(\vec{x'})\overline{d'}} \overline{e}} \qquad \text{(C:IN)}\quad \frac{}{\overline{\mathtt{1}} \xrightarrow{c} \overline{c}}$$

$$\text{(C:SYNC)}\quad \frac{c \Vdash_C \exists\vec{y}(d\{\vec{t}/\vec{x}\} \otimes e) \quad \vec{y} \cap fv(c, d, P) = \emptyset \qquad \mathbf{mgc}\big(c, \exists\vec{y}(d\{\vec{t}/\vec{x}\} \otimes e)\big)}{\overline{c} \parallel \forall\vec{x}(d \rightarrow P) \xrightarrow{\tau} \exists\vec{y}.\,(P\{\vec{t}/\vec{x}\} \parallel \overline{e})}$$

$$\text{(C:COMP)}\quad \frac{P \xrightarrow{\alpha} P' \quad ev(\alpha) \cap fv(Q) = \emptyset}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \text{(C:SUM)}\quad \frac{P \parallel G_i \xrightarrow{\alpha} P' \quad i \in \{1, 2\}}{P \parallel G_1 + G_2 \xrightarrow{\alpha} P'}$$

$$\text{(C:EXT)}\quad \frac{P \xrightarrow{(\vec{x})\overline{c}} Q}{\exists y.\, P \xrightarrow{(y\vec{x})\overline{c}} Q} \qquad \text{(C:RES)}\quad \frac{P \xrightarrow{\alpha} P' \quad y \notin fv(\alpha)}{\exists y.\, P \xrightarrow{\alpha} \exists y.\, P'}$$

$$\text{(C:CONG)}\quad \frac{P \equiv_l P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv_l Q}{P \xrightarrow{\alpha} Q}$$

Figure 4. Labeled Transition System (LTS) for `lcc` processes.

Intuitively, the $\mathbf{mgc}$ predicate allows us to refer formally to decompositions of constraints that do not "lose" or "forget" information. This is essential in the presence of linear constraints.

We comment on the rules of Fig. 4. Rule (C:OUT) formalizes asynchronous tells: using $\mathbf{mgc}$, the sent constraint is decomposed in two parts: the first one is actually sent (as recorded in the label); the second part is kept as a continuation. Rule (C:IN) asynchronously receives a constraint; it represents the separation between observing an output and its (asynchronous) reception, which is not directly observable. Rule (C:SYNC) formalizes the synchronization between a tell (i.e., an output) and an ask. As before, the constraint mentioned in the tell is decomposed using $\mathbf{mgc}$: here the first part is used (consumed) to "trigger" the processes guarded by the ask, while the second part is the remaining continuation. Rules (C:COMP), (C:SUM) are self-explanatory. Rules (C:EXT) and (C:RES) formalize hiding. Finally, rule (C:CONG) closes transitions under structural congruence (cf. Def. 3.8).

Weak transitions are standardly defined: we write $P \stackrel{\tau}{\Longrightarrow} Q$ iff $(P \xrightarrow{\tau}{}^{*} Q)$ and $P \stackrel{\alpha}{\Longrightarrow} Q$ iff $(P \xrightarrow{\tau}{}^{*} P' \xrightarrow{\alpha} P'' \xrightarrow{\tau}{}^{*} Q)$. Reduction $P \rightarrow_l Q$ is defined as $P \equiv_l \xrightarrow{\tau} \equiv_l Q$. We write $\rightarrow_l^k$ to indicate k consecutive reductions ($k \geq 1$).

To reason about encoding correctness, we exploit *observational equivalences* for `lcc` processes. The following auxiliary definition gives the set of $\mathcal{D}$-accessible constraints, as defined in [13].

Definition 3.10 (**$\mathcal{D}$-accessible constraints**). *Let $\mathcal{D} \subset \mathcal{C}$, where $\mathcal{C}$ is the set of all constraints. The observables of an* `lcc` *process P are the set of all $\mathcal{D}$-accessible constraints defined as follows:*

$$\mathcal{O}^{\mathcal{D}}(P) = \{(\exists x.c) \in \mathcal{D} \,|\, \exists P'.\, P \stackrel{\tau}{\Longrightarrow} \exists x.(P' \parallel \overline{c})\}$$

We now define a bisimulation relation for `lcc` processes. Let $\mathcal{D}, \mathcal{E} \subseteq \mathcal{C}$. We say that an action (label) is $\mathcal{DE}$-relevant for a process P if it is either a silent action τ, or an input action in $\mathcal{E}$, or an output action $(\vec{x})\overline{c}$ with $\vec{x} \cap fv(P) = \emptyset$ and $\exists\vec{x}.c \in \mathcal{D}$.

Definition 3.11 (**$\mathcal{DE}$-bisimulation**). *Let $\mathcal{D}, \mathcal{E} \subseteq \mathcal{C}$, where $\mathcal{C}$ is the set of all constraints. Then we say that a symmetric relation $\mathcal{R}$ is a $\mathcal{DE}$-bisimulation if for all processes P, P', Q and for all labels α such that $P\mathcal{R}Q$, $P \xrightarrow{\alpha} P'$ and α is a $\mathcal{DE}$-relevant action for Q, there exists Q' s.t $Q \stackrel{\alpha}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$. The largest $\mathcal{DE}$-bisimulation is called $\mathcal{DE}$-bisimilarity and is denoted $\approx_{\mathcal{DE}}$*

We will assume $\mathcal{D} = \mathcal{E} = \mathcal{C}$; bisimilarity will be denoted by $\approx$.

4. Encoding sπ in `lcc`

We now introduce an encoding of sπ into `lcc` and its associated correctness results. We first present the abstract notion of encoding that we consider (§ 4.1). Then, we describe the encoding (§ 4.2) and in § 4.3 and § 4.4 we establish its correctness (see Cor. 4.13).

4.1 The Notion of Encoding

Our notion of encoding is inspired by that proposed by Gorla [12].

Definition 4.1 (Calculi and Translations). *Assume a countably infinite set of variables* V. *A* calculus $\mathcal{L}$ *is a tuple* $\langle \mathsf{P}, \longrightarrow, \approx \rangle$, *where* P *is a set of processes,* $\longrightarrow$ *denotes an operational semantics, and* $\approx$ *is a behavioral equality on* P.
Given calculi $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \approx_s \rangle$ *and* $\mathcal{L}_t = \langle \mathsf{P}_t, \longrightarrow_t, \approx_t \rangle$, *a* translation *from* $\mathcal{L}_s$ *into* $\mathcal{L}_t$ *is a tuple* $\langle \llbracket \cdot \rrbracket, \psi_{\llbracket \cdot \rrbracket} \rangle$, *where*
$\llbracket \cdot \rrbracket : \mathsf{P}_s \rightarrow \mathsf{P}_t$, *denotes a process mapping, and* $\varphi_{\llbracket \cdot \rrbracket} : \mathsf{V}_s \rightarrow \mathsf{V}_t$ *is injective function denoting a renaming policy for* $\llbracket \cdot \rrbracket$.

Calculi $\mathcal{L}_s$ and $\mathcal{L}_t$ are the *source* and *target* of the translation, respectively. We assume that their semantics is based on a reduction relation. We write $\Longrightarrow_s$ to denote a the reflexive, transitive closure of $\longrightarrow_s$ (and similarly for $\longrightarrow_t$). The renaming policy defines the way in which the translation maps variables from the source to the target calculus; this is useful to express that the process mapping does not depend on specific substitutions. We are interested in *encodings*, i.e., translations which enjoy certain *encodability criteria*.

Definition 4.2 (Encoding). *Let* $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \approx_s \rangle$ *and* $\mathcal{L}_t = \langle \mathsf{P}_t, \longrightarrow_t, \approx_t \rangle$ *be calculi in the sense of Def. 4.1. A translation* $\langle \llbracket \cdot \rrbracket, \varphi_{\llbracket \cdot \rrbracket} \rangle$ *of* $\mathcal{L}_s$ *into* $\mathcal{L}_t$ *is an* encoding *if it satisfies:*

1. ***Name invariance****: Let a be a variable in $\mathcal{L}_s$. Then, for all $S \in \mathsf{P}_s$ and every possible substitution σ then $\llbracket S\sigma \rrbracket = \llbracket S \rrbracket \sigma'$, with $\varphi_{\llbracket \cdot \rrbracket}(\sigma(x)) = \sigma'(\varphi_{\llbracket \cdot \rrbracket}(x))$, for every variable x in P_s.*
2. ***Compositionality*** *(with respect to parallel and restriction):*

$$\begin{aligned} \llbracket \mathsf{res}_s(\vec{x}, P) \rrbracket &= \mathsf{res}_t(\vec{x}, \llbracket P \rrbracket) \\ \llbracket \mathsf{par}_s(P, Q) \rrbracket &= \mathsf{par}_t(\llbracket P \rrbracket, \llbracket Q \rrbracket) \end{aligned}$$

   where $\mathsf{res}_s(\cdot,\cdot)$ *and* $\mathsf{par}_s(\cdot,\cdot)$ *(resp.* $\mathsf{res}_t(\cdot,\cdot)$ *and* $\mathsf{par}_t(\cdot,\cdot)$*) denote operators for restriction and parallel composition on processes in* P_s *(resp.* P_t*).*
3. ***Operational correspondence****, i.e., it is sound and complete:*
   (a) ***Soundness:*** *For all $S \in \mathsf{P}_s$, if $S \longrightarrow_s S'$, there exist $T \in \mathsf{P}_t$ such that $\llbracket S \rrbracket \Longrightarrow_t T$ and $T \approx_t \llbracket S' \rrbracket$.*
   (b) ***Completeness:*** *For all $S \in \mathsf{P}_s$ and $T \in \mathsf{P}_t$, if $\llbracket S \rrbracket \Longrightarrow_t T$, there exist S' such that $S \longrightarrow_s S'$ and $T \approx_t \llbracket S' \rrbracket$.*

Intuitively, *name invariance* ensures that translations respect the declared renaming policy. *Compositionality* ensures that the translation of a process is defined in terms of translations of its subprocesses. As we consider very different calculi, we focus on compositionality in terms of parallel composition and restriction. *Operational correspondence* ensures that a translated process in the target calculus preserves (up to behavioral equivalence) the behavior of its associated source processes (*soundness*). The converse is *completeness*: the behavior of a translated (target) process should correspond to some behavior of the source process.

Encodability criteria can be *static* or *dynamic*. Static criteria refer to structural properties of the translation; dynamic criteria relates the behavior of a target process and that of its corresponding source process. Name invariance and compositionality are static criteria; operational correspondence is a dynamic criterion.

4.2 Translating sπ into `lcc`

We move to consider sπ and `lcc` as source and target languages in a translation. We first define the constraint system that we will use:

$$\Sigma \stackrel{\texttt{def}}{=} \; \mathit{in}(x,y) \;\big|\; \mathit{out}(x,y) \;\big|\; \mathit{sel}(x,l) \;\big|\; \mathit{br}(x,l) \;\big|\; \mathit{covar}(x,y)$$

Figure 5. Session constraint system: Predicates

$$\begin{aligned}
\llbracket \overline{x}\, v.P \rrbracket &= \overline{\mathit{out}(x,v)} \parallel \forall z\big(\mathit{in}(z,v) \otimes \{x{:}z\} \to \llbracket P \rrbracket\big)\\
\llbracket x(y).P \rrbracket &= \forall y,w\big(\mathit{out}(w,y) \otimes \{w{:}x\} \to \overline{\mathit{in}(x,y)} \parallel \llbracket P \rrbracket\big)\\
\llbracket x \triangleleft l.P \rrbracket &= \overline{\mathit{sel}(x,l)} \parallel \forall z\big(\mathit{br}(z,l) \otimes \{x{:}z\} \to \llbracket P \rrbracket\big)\\
\llbracket x \triangleright \{l_i{:}P_i\}_{i\in I} \rrbracket &= \forall l,w\big(\mathit{sel}(w,l) \otimes \{w{:}x\} \to\\
&\qquad \overline{\mathit{br}(x,l)} \parallel \prod_{1\le i\le n} \forall\epsilon(l = l_i \to \llbracket P_i \rrbracket)\big)\\
\llbracket v?\,(P){:}(Q) \rrbracket &= \forall\epsilon(v = \mathtt{tt} \to \llbracket P \rrbracket) \parallel \forall\epsilon(v = \mathtt{ff} \to \llbracket Q \rrbracket)\\
\llbracket P \mid Q \rrbracket &= \llbracket P \rrbracket \parallel \llbracket Q \rrbracket\\
\llbracket \mathbf{0} \rrbracket &= \overline{\mathtt{1}}\\
\llbracket (\boldsymbol{\nu} xy)P \rrbracket &= \exists x,y.\,(!\overline{\{x{:}y\}} \parallel \llbracket P \rrbracket)\\
\llbracket *\, x(y).P \rrbracket &= !\, \llbracket x(y).P \rrbracket
\end{aligned}$$

In $\llbracket \overline{x}\, v.P \rrbracket$ and $\llbracket x \triangleleft l.P \rrbracket$, we assume $z \notin fv(P)$.
Also, we assume $w, z \notin fv(P)$ in $\llbracket x(y).P \rrbracket$ and $\llbracket x \triangleright \{l_i{:}P_i\}_{i\in I} \rrbracket$.

Figure 6. Translation from $\mathtt{s}\pi$ to `lcc`.

Definition 4.3 (Session Constraint System). *Consider the tuple $\langle \mathcal{C}, \Sigma, \Vdash_C \rangle$ where $\mathcal{C}$ is the set of all constraints obtained by using linear logic operators !, $\otimes$ and $\exists$ over the predicates of Σ (Fig. 5) and $\Vdash_C$ is given by the usual deduction rules for linear logic with syntactic equality.*

We will use the predicates in Fig. 5 to model the synchronization of $\mathtt{s}\pi$ processes (see below). The following notation is useful:

Notation 4.4 (Co-variables). The constraint $\mathit{covar}(x,y)$, used to denote a pair of co-variables, will be written $\{x{:}y\}$.

We may now formally introduce the translation:

Definition 4.5 ($\mathtt{s}\pi$ into `lcc`). *We define the translation from $\mathtt{s}\pi$ programs into `lcc` processes as the tuple $\langle \llbracket\cdot\rrbracket, \varphi_{\llbracket\cdot\rrbracket} \rangle$, where:*

(a) $\llbracket\cdot\rrbracket$ is the process mapping defined in Fig. 6.
(b) $\varphi_{\llbracket\cdot\rrbracket}$ is defined as $\varphi_{\llbracket\cdot\rrbracket}(x) = x$, i.e., each variable in $\mathtt{s}\pi$ is mapped to the same variable in `lcc`.

Some intuitions on the mapping $\llbracket\cdot\rrbracket$ in Fig. 6. follow. We use predicates $\mathit{in}(\cdot)$ and $\mathit{br}(\cdot)$ in Fig. 5 to represent acknowledgment messages used for synchronization. This is a simple way of dealing with the fact that $\mathtt{s}\pi$ is a synchronous language, whereas `lcc` processes follow an asynchronous communication discipline. Note that the translation of communication prefixes (output, input, selection, branching) uses a constraint $\{x{:}y\}$ to denote the fact that x and y are co-variables. The persistent availability of such a constraint is defined by the translation of process $(\boldsymbol{\nu} xy)P$.

As an example, consider the translation of the $\mathtt{s}\pi$ program featuring an input-output synchronization (the translation of selection and branching prefixes is quite similar). Let P be the $\mathtt{s}\pi$ redex

$$(\boldsymbol{\nu} xy)(\overline{x}\, v.P_1 \mid y(u).P_2)$$

The `lcc` process $\llbracket P \rrbracket$ is defined as:

$$\begin{aligned}
&\exists x,y.\ \big(!\overline{\{x{:}y\}} \parallel \overline{\mathit{out}(x,v)} \parallel \forall z(\mathit{in}(z,v) \otimes \{x{:}z\} \to \llbracket P_1 \rrbracket)\\
&\qquad \parallel \forall u,w(\mathit{out}(w,u) \otimes \{w{:}y\} \to \overline{\mathit{in}(y,u)} \parallel \llbracket P_2 \rrbracket)\big)
\end{aligned}$$

Omitting some unimportant substitutions, $\llbracket P \rrbracket$ intuitively behaves as follows. Observe how process $\overline{\mathit{out}(x,v)}$, added by the translation of output, is meant to interact with the abstraction in the translation of input. The encoding of restriction provides unlimited copies of the co-variable constraint $\{x{:}y\}$; this suffices to trigger the process $\overline{\mathit{in}(y,v)} \parallel \llbracket P_2 \rrbracket\{^{v}\!/_{u}\}$, containing the continuation of the input. Once that occurs, a similar pattern in the reverse direction is performed: constraint $\mathit{in}(y,v)$ and the co-variable constraint will trigger the continuation of the output, denoted $\llbracket P_1 \rrbracket$. This completes the declarative representation of the reduction from P.

We move on to establish *correctness* for this translation, i.e., to establish that it adheres to the notion of encoding in Def. 4.2.

4.3 Translation Correctness (1): Static Properties

We now show that $\llbracket\cdot\rrbracket$ is name invariant with respect to the renaming policy in Def. 4.5(b). This proves condition Def. 4.2 (1):

Theorem 4.6 (Name invariance for $\llbracket\cdot\rrbracket$). *Let P, σ, and x be a typable $\mathtt{s}\pi$ process, a substitution satisfying the renaming policy for $\llbracket\cdot\rrbracket$ (Def. 4.5(b)), and a variable in $\mathtt{s}\pi$, resp. Then $\llbracket P\sigma \rrbracket = \llbracket P \rrbracket\sigma'$, with $\varphi_{\llbracket\cdot\rrbracket}(\sigma(x)) = \sigma'(\varphi_{\llbracket\cdot\rrbracket}(x))$ and $\sigma = \sigma'$.*

To simplify the presentation of the semantic properties, we define the usual notion of *evaluation contexts* for $\mathtt{s}\pi$.

Definition 4.7 (Evaluation Contexts ($\mathtt{s}\pi$)). *The syntax of evaluation contexts in $\mathtt{s}\pi$ is given by the following grammar, where P is an $\mathtt{s}\pi$ process:*

$$E ::= \cdot \;\big|\; E \mid P \;\big|\; P \mid E \;\big|\; (\boldsymbol{\nu} xy)(E)$$

An (evaluation) context is a process with a "hole", denoted '$\cdot$'. Given an evaluation context $E[\cdot]$, we write $E[P]$ to denote the $\mathtt{s}\pi$ process that results from filling in the occurrences of the hole with process P. We will write $C[\cdot]$ when referring to evaluation contexts with outermost restrictions only, e.g., $(\boldsymbol{\nu} xy)(\,\cdot\,)$.

We now prove compositionality of $\llbracket\cdot\rrbracket$ with respect to restriction and parallel composition operator, as in Def. 4.2 (2).

Theorem 4.8 (Compositionality of $\llbracket\cdot\rrbracket$). *Let P and $E[\cdot]$ be a typable $\mathtt{s}\pi$ process and an $\mathtt{s}\pi$ evaluation context as in Def. 4.7, respectively. Then we have: $\llbracket E[P] \rrbracket = \llbracket E \rrbracket\big[\llbracket P \rrbracket\big]$.*

We have thus established that our translation satisfies static encoding criteria; we now investigate operational correspondence, a dynamic encodability criterion.

4.4 Translation Correctness (2): Operational Correspondence

One important issue to be addressed with $\llbracket\cdot\rrbracket$ is the intrinsic non-determinism of $\mathtt{s}\pi$. This is crucial since it is desirable that our translation captures the non-deterministic behavior of processes with unrestricted channels (e.g., a server communicating with multiple clients). This class of processes is not captured in our previous work [19]. Consider the $\mathtt{s}\pi$ program below, *not encodable in [19]*:

$$Q = (\boldsymbol{\nu} xy)(\overline{x}\, v_1.P_1 \mid \overline{x}\, v_2.P_2 \mid y(z).R) \tag{1}$$

which is typable in [27] with a context $\Gamma = \{x : \mu a.\mathit{un}!\mathit{bool}.T, y : \mathit{lin}?\mathit{bool}.U\}$. We have either

$$\begin{aligned}
Q &\quad\to_\pi\quad (\boldsymbol{\nu} xy)(P_1 \mid \overline{x}\, v_2.P_2 \mid R\{^{v_1}\!/_{z}\}) = Q_1\\
Q &\quad\to_\pi\quad (\boldsymbol{\nu} xy)(\overline{x}\, v_1.P_1 \mid P_2 \mid R\{^{v_2}\!/_{z}\}) = Q_2
\end{aligned}$$

Now consider the `lcc` process $\llbracket Q \rrbracket$:

$$\begin{aligned}
\exists x,y.\big(\,!\,\overline{\{x{:}y\}} &\parallel \overline{\mathit{out}(x,v_1)} \parallel \forall z(\mathit{in}(z,v_1) \otimes \{x{:}z\} \to \llbracket P_1 \rrbracket) \parallel\\
&\overline{\mathit{out}(x,v_2)} \parallel \forall z(\mathit{in}(z,v_2) \otimes \{x{:}z\} \to \llbracket P_2 \rrbracket) \parallel\\
&\forall z,w(\mathit{out}(w,z) \otimes \{w{:}y\} \to \overline{\mathit{in}(y,z)} \parallel \llbracket R \rrbracket)\big)
\end{aligned}$$

$$
\begin{aligned}
\llbracket Q\rrbracket \equiv_l{} & \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \overline{out(x, v_1) \otimes out(x, v_2)} \parallel \\
& \qquad \forall z(in(z, v_1) \otimes \{x{:}z\} \rightarrow \llbracket P_1\rrbracket) \parallel \\
& \qquad \forall z(in(z, v_2) \otimes \{x{:}z\} \rightarrow \llbracket P_2\rrbracket) \parallel \\
& \qquad \forall z, w(out(w, z) \otimes \{w{:}y\} \rightarrow \overline{in(y, z)} \parallel \llbracket R\rrbracket)\big) \\
\rightarrow_l{} & \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \overline{out(x, v_2)} \parallel \\
& \qquad \forall z(in(z, v_1) \otimes \{x{:}z\} \rightarrow \llbracket P_1\rrbracket) \parallel \overline{in(y, v_1)} \parallel \\
& \qquad \forall z(in(z, v_2) \otimes \{x{:}z\} \rightarrow \llbracket P_2\rrbracket) \parallel \llbracket R\rrbracket\{v_1, x/z, w\}\big) \\
\equiv_l{} & \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \overline{out(x, v_2) \otimes in(y, v_1)} \parallel \\
& \qquad \forall z(in(z, v_1) \otimes \{x{:}z\} \rightarrow \llbracket P_1\rrbracket) \parallel \\
& \qquad \forall z(in(z, v_2) \otimes \{x{:}y\} \rightarrow \llbracket P_2\rrbracket) \parallel \llbracket R\rrbracket\{v_1, x/z, w\}\big) \\
\rightarrow_l{} & \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \overline{out(x, v_2)} \parallel \llbracket P_1\{y/z\}\rrbracket \parallel \\
& \qquad \forall z((in(z, v_2) \otimes \{z{:}y\}) \rightarrow \llbracket P_2\rrbracket) \parallel \llbracket R\rrbracket\{v_1, x/z, w\}\big) \\
\equiv_l{} & \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \llbracket P_1\{y/z\}\rrbracket \parallel \overline{(out(x, v_2)} \parallel \\
& \qquad \forall z((in(z, v_2) \otimes \{x{:}z\}) \rightarrow \llbracket P_2\rrbracket) \parallel \llbracket R\rrbracket\{v_1, x/z, w\}\big)
\end{aligned}
$$

Figure 7. Evolution of the lcc translation of program (1) (§ 4.4).

One can show that $\llbracket Q\rrbracket$ reaches a state in which only one of the outputs will interact with the input process; Fig. 7 details this evolution. Given the definition of $\llbracket\cdot\rrbracket$, we may see that the resulting process is the translation for Q_1 above. This justifies the use of a calculus related to linear logic as the basis for the presented translation, since it allows us to represent these forms of non-determinism (not considered in our previous work [19]). Here non-determinism is in the fact that $\llbracket Q\rrbracket$ may also evolve into $\llbracket Q_2\rrbracket$.

Observe the process obtained before the continuation $\llbracket P_1\rrbracket$ is executed, i.e., $\forall z((in(z, v_1) \otimes \{x{:}z\}) \rightarrow \llbracket P_1\rrbracket)$. We will give an informative statement of operational correspondence by precisely characterizing these *intermediate processes*.

We require some auxiliary results and definitions. The following lemma establishes the shape of a well-formed program. We say that a process is a *pre-redex* if it is prefixed at some variable, i.e., it does not contain parallel composition at the top-level. Note that the composition of two pre-redexes may constitute a redex (cf. § 3.1).

Lemma 4.9 (Translated form of a program). *Let P be a well-typed sπ program ($\vdash P$) (Not. 3.5), then*

$$\llbracket P\rrbracket \equiv_l \exists\vec{x}, \vec{y}.(\llbracket R_1\rrbracket \parallel \ldots \parallel \llbracket R_n\rrbracket \parallel V)$$

where $n \geq 1$, $V = !\overline{\{x_1{:}y_1\}} \parallel \ldots \parallel !\overline{\{x_n{:}y_n\}}$, $x_1, \ldots, x_n \in \vec{x}$, and $y_1, \ldots, y_n \in \vec{y}$. Note that each $R_i, 1 \leq i \leq n$ is a pre-redex.

Definition 4.10 (Continuation processes). *Let P be an sπ process such that $P \equiv_\pi (\boldsymbol{\nu}\vec{x}\vec{y})(\overline{x_i}\, v.Q \mid R)$ or $P \equiv_\pi (\boldsymbol{\nu}\vec{x}\vec{y})(x_i \triangleleft l.Q \mid R)$, for some Q, R, $\vec{x}$, $\vec{y}$, l. Assume $x_i \in \vec{x}$, $y_i \in \vec{y}$ are co-variables. The* continuation process *of P, denoted $\llparenthesis P\rrparenthesis_{y_i}$, is defined as follows:*

1. *If $P \equiv_\pi (\boldsymbol{\nu}\vec{x}\vec{y})(\overline{x_i}\, v.Q \mid R)$ then*
   $\llparenthesis P\rrparenthesis_{y_i} = \forall z((in(y_i, v) \otimes \{z{:}y_i\}) \rightarrow \llbracket Q\rrbracket)$.
2. *If $P \equiv_\pi (\boldsymbol{\nu}\vec{x}\vec{y})(x_i \triangleleft l.Q \mid R)$ then*
   $\llparenthesis P\rrparenthesis_{y_i} = \forall z((br(y_i, l) \otimes \{z{:}y_i\}) \rightarrow \llbracket Q\rrbracket)$.

We write $\llparenthesis P\rrparenthesis$ when the co-variable y_i is unimportant.

We may now define:

Definition 4.11 (Intermediate processes). *Let P be a typable sπ program. Consider its encoded form (Lem. 4.9), given as follows:*

$$\llbracket P\rrbracket = \llbracket C\rrbracket\big[\llbracket R_1\rrbracket, \ldots, \llbracket R_i\rrbracket, \ldots, \llbracket R_n\rrbracket\big]$$

with $1 \leq i \leq n$. Let S be an lcc *process such that*

$$S = \llbracket C\rrbracket\big[\llbracket R_1\rrbracket, \ldots, \llparenthesis R_i\rrparenthesis, \ldots, \llbracket R_n\rrbracket\big], 1 \leq i \leq n$$

We say S is an intermediate process *of $\llbracket P\rrbracket$, denoted $S \in \wr\llbracket P\rrbracket\int$, if there exist S' and S'' such that $\llbracket P\rrbracket \equiv_l S' \xrightarrow{\tau} S'' \equiv_l S$.*

The previous definitions give us an idea of how reductions are represented by our translation. We may see that encoded redexes must first reach an intermediate process. This intermediate process can be related to a state where the message that triggers the continuation of the output (selection) process has not yet been received. Intermediate processes are key to state the operational correspondence theorem below, which is a basis for ensuring dynamic properties for the transference of reasoning techniques from lcc to sπ:

Theorem 4.12 (Operational Correspondence for $\llbracket\cdot\rrbracket$). *Let $\llbracket\cdot\rrbracket$ be the translation in Def. 4.5. Also, let P, Q be well-typed sπ programs and R, S be* lcc *processes. Then:*

1. **Soundness:** *If $P \rightarrow_\pi Q$ then either:*
   a) $\llbracket P\rrbracket \rightarrow_l R$, such that $R \approx \llbracket Q\rrbracket$
   b) (or) $\llbracket P\rrbracket \equiv_l S' \rightarrow_l^2 R' \equiv_l R$, for some R', S', R such that $R \approx \llbracket Q\rrbracket$.
2. **Completeness:** *If $\llbracket P\rrbracket \rightarrow_l S$. Then either:*
   a) $P \rightarrow_\pi Q$ for some Q and $\llbracket Q\rrbracket \approx S$.
   b) (or) $S \in \wr\llbracket P\rrbracket\int$ and for some S' we have that $S \rightarrow_l S'$. Also, $P \rightarrow_\pi Q$ for some Q and $\llbracket Q\rrbracket \approx S'$.

Informally, cases (a) capture reduction of conditional expressions; cases (b) capture other kinds of reduction.

We now state the main result of this section: our translation is a correct encoding, as it satisfies the static and dynamic criteria in Def. 4.2. It is a consequence of Theorems 4.6, 4.8, and 4.12.

Corollary 4.13. *Translation $\langle\llbracket\cdot\rrbracket, \varphi_{\llbracket\cdot\rrbracket}\rangle$ is an encoding (cf. Def. 4.2).*

5. A Type System for lcc

We will now introduce a type system for lcc that limits the power of abstractions, and establish its main properties. This additional control relies on a generalization of abstractions, motivated next.

5.1 Linear Abstractions with Local Information

We consider a variant of lcc in which abstractions are generalized so as to account for *local information*:

$$\forall\vec{x}(d\,;e \rightarrow P)$$

Intuitively, d is a piece of local information used jointly with e to trigger P. Abstractions with local information refine the abstractions in [13], which act on the global information posted in the store. This might be an issue when dealing with processes that appeal to information private to them in order to perform their (publicly visible) behavior. We extend Fig. 4 with the following rule:

$$
\textsc{(C:SynLoc)}\quad
\frac{\begin{array}{c} c \otimes d \Vdash_C \exists\vec{y}.(e\{\vec{t}/\vec{x}\} \otimes f) \quad \vec{y} \cap fv(c, d, e, P) = \emptyset \\ \mathbf{mgc}\big(c \otimes d, \exists\vec{y}.(e\{\vec{t}/\vec{x}\} \otimes f)\big) \quad c \otimes d \Vdash_C \mathtt{0} \Rightarrow c \Vdash_C \mathtt{0} \end{array}}{\overline{c} \parallel \forall\vec{x}(d\,;e \rightarrow P) \xrightarrow{\tau} \exists\vec{y}.\,(P\{\vec{t}/\vec{x}\} \parallel \overline{f})}
$$

The idea is to infer e using d without publishing d to the store. Examples of local information are (private) keys used in protocols for secure communications. Premise $c \otimes d \Vdash_C \mathtt{0} \Rightarrow c \Vdash_C \mathtt{0}$ ensures that only local assumptions which do not conflict with the information in the global store are allowed. The use of abstractions using local information will be illustrated in § 6.

5.2 Type System: Motivation

The encoding of $\mathsf{s}\pi$ into `lcc` introduced in § 4 relies critically on abstractions to represent synchronizations in $\mathsf{s}\pi$, as required to encode session communications (including scope extrusions) and their associated continuations. Unfortunately, the abstraction mechanism in `lcc` is overly powerful for modeling scope extrusion, in the sense that abstraction can represent scenarios not possible in $\mathsf{s}\pi$ by combining name passing and restriction. Precisely, the private character of synchronizations on restricted channels is not respected by abstraction-based encodings. We illustrate this anomaly using a simple example. Consider the $\mathsf{s}\pi$ process:

$$S = (\boldsymbol{\nu} xy)(\overline{x}\, v.P_x \mid y(z).Q_y) \mid R \tag{2}$$

Under the assumption that $\mathsf{fn}(P,Q) \cap \mathsf{fn}(R) = \emptyset$, the restriction $(\boldsymbol{\nu} xy)$ ensures that communications between endpoints x and y are private, i.e., they cannot be interfered by some external process. In particular, we have that R cannot get ahold of v in the reduction

$$S \rightarrow_\pi (\boldsymbol{\nu} xy)(P_x \mid Q_y\{{}^{v}\!/\!{}_{z}\}) \mid R \tag{3}$$

Unfortunately, the privacy guarantees offered by restriction in $\mathsf{s}\pi$ do not extend to `lcc`, which seriously hinders one of the main assumptions in session-based concurrency. Consider the `lcc` process

$$[\![(\boldsymbol{\nu} xy)(\overline{x}\, v.P_x \mid y(z).Q_y)]\!] \parallel A \tag{4}$$

where A could represent a malicious attacker that spies the communication endpoint x, y for the benefit of some process Spy:

$$A = \forall y, w(\mathtt{tt}\,\boldsymbol{;}\,(out(w,y) \otimes \{x{:}w\}) \rightarrow (\overline{in(x,y)} \parallel Spy))$$

Process A abstracts both the endpoint and the message in transit, performs an operation, and signals a correct input. It is easy to see that in a context including A, process $[\![(\boldsymbol{\nu} xy)(\overline{x}\, v.P_x \mid y(z).Q_y)]\!]$ could synchronize according to the session, but could also (wrongly) interact with A, thus breaching session privacy. Thus, the (deterministic) reduction in (3) can no longer be ensured when $\mathsf{s}\pi$ processes are compiled down into `lcc`.

Note that this anomaly is not particular of our encoding $[\![\cdot]\!]$; rather, it affects all `lcc` processes that use abstractions to synchronize input-like processes. Scope extensions as the one possible in (4) are clearly not possible in $\mathsf{s}\pi$, and we must limit the power of abstractions so as to preserve the very nature of the restriction operator in $\mathsf{s}\pi$. Intuitively, this means that the privacy of session endpoints must be explicitly programmed at the declarative level of `lcc`, relying on some extra mechanism that limits abstractions.

To this end, we rely on a simple *typing discipline*, built upon the approach in [15] (where the focus is in `utcc` and session-based concurrency is not addressed). Our type system admits only abstractions which adhere to a precisely defined unrestricted/restricted policy. Intuitively, this means that instead of abstracting from constraints with a single, unrestricted set, we distinguish variables between two sorts: one denoting *unrestricted* (i.e., public) variables/data, and another denoting *restricted* (i.e., privacy-sensitive, non-abstractable) variables/data. This can be seen as an access control mechanism for `lcc` abstractions.

A well-typed `lcc` process in our type system is a process in which all abstractions $\forall \vec{x}(d\,\boldsymbol{;}\,e \rightarrow P)$ are such that e is a *secure pattern*, i.e., it respects the sorting policy and does not concern non-abstractable variables.

The type system is defined in general terms; one application is our encoding of $\mathsf{s}\pi^+$ into `lcc`; see § 6. In this case, the sorting policy applies to the predicates used to represent synchronizations. This way, e.g., we will assume a signature where $out(x,y)$ is a function with x restricted and y unrestricted, and in which $\{x{:}y\}$ is a function having both x and y restricted names. This allows us to distinguish process $[\![(\boldsymbol{\nu} xy)(\overline{x}\, v.P_x \mid y(z).Q_y)]\!]$ from process $[\![(\boldsymbol{\nu} xy)(\overline{x}\, v.P_x \mid y(z).Q_y)]\!] \parallel A$: while the former is well-typed, the latter is not (see also Example 5.3).

$$\text{(L:True)}\ \frac{}{\cdot;\cdot \vdash_{\bullet} \mathtt{1}} \qquad \text{(L:False)}\ \frac{}{\cdot;\cdot \vdash_{\bullet} \mathtt{0}} \qquad \text{(L:Assoc-l)}\ \frac{\Delta;\Theta \vdash_{\bullet} (c \otimes d) \otimes e}{\Delta;\Theta \vdash_{\bullet} c \otimes (d \otimes e)}$$

$$\text{(L:Comm)}\ \frac{\Delta;\Theta \vdash_{\bullet} c \otimes d}{\Delta;\Theta \vdash_{\bullet} d \otimes c} \qquad \text{(L:Pred)}\ \frac{\Delta = var(\vec{t_1}\,) \cup res(\vec{t_2}\,) \quad \Theta = unr(\vec{t_2}\,) \backslash \Delta}{\Delta;\Theta \vdash_{\bullet} \gamma(\vec{t_1};\vec{t_2})}$$

$$\text{(L:Comb)}\ \frac{\Delta_1;\Theta_1 \vdash_{\bullet} c \quad \Delta_2;\Theta_2 \vdash_{\bullet} d \quad (\Delta_1 \cap \Theta_2) = (\Delta_2 \cap \Theta_1) = \emptyset}{\Delta_1, \Delta_2;\Theta_1, \Theta_2 \vdash_{\bullet} c \otimes d}$$

$$\text{(L:Exist)}\ \frac{\Delta;\Theta \vdash_{\bullet} c}{\Delta;\Theta \vdash_{\bullet} \exists \vec{x}.c} \qquad \text{(L:Bang)}\ \frac{\Delta;\Theta \vdash_{\bullet} c}{\Delta;\Theta \vdash_{\bullet} !c}$$

$$\text{(L:Abs)}\ \frac{\vdash_{\diamond} P \quad \Delta;\Theta \vdash_{\bullet} c \quad \vec{x} \subseteq dom(\Theta) \setminus fv(d)}{\vdash_{\mathbf{A}} \forall \vec{x}(d\,\boldsymbol{;}\,c \rightarrow P)}$$

$$\text{(L:Sum)}\ \frac{\vdash_{\mathbf{A}} G_1 \quad \vdash_{\mathbf{A}} G_2}{\vdash_{\mathbf{A}} G_1 + G_2} \qquad \text{(L:Guard)}\ \frac{\vdash_{\mathbf{A}} G}{\vdash_{\diamond} G}$$

$$\text{(L:Tell)}\ \frac{c \in \mathcal{C}}{\vdash_{\diamond} \overline{c}} \qquad \text{(L:Par)}\ \frac{\vdash_{\diamond} P_1 \quad \vdash_{\diamond} P_2}{\vdash_{\diamond} P_1 \parallel P_2} \qquad \text{(L:Repl)}\ \frac{\vdash_{\diamond} P}{\vdash_{\diamond} !P} \qquad \text{(L:Local)}\ \frac{\vdash_{\diamond} P}{\vdash_{\diamond} \exists \vec{x}.\, P}$$

Figure 8. Typing rules for `lcc`. Rule (L:Assoc-r) is omitted.

5.3 The Typing System

The typing rules for secure patterns/processes are defined in Fig. 8. For simplicity, we assume that patterns are conjunctions of predicates applied to terms over the function signature. We consider two environments, Δ and Θ: while Δ is the set of variables used restricted, Θ is the set of variables used unrestricted. We employ three functions on terms: $unr(t)$, $res(t)$, and $var(t)$, yielding, respectively, the variables appearing unrestricted in t according to the sorting, the variables appearing restricted in t, and all variables appearing in t. Formally, these functions are given by:

$$\begin{array}{rcl} unr(x) & = & res(x) = var(x) = \{x\} \ \ (x \text{ is a variable}) \\ unr(\gamma(\vec{t_1};\vec{t_2})) & = & unr(\vec{t_2}) \\ res(\gamma(\vec{t_1};\vec{t_2})) & = & res(\vec{t_1}) \\ var(\gamma(\vec{t_1};\vec{t_2})) & = & var(\vec{t_1}) \cup var(\vec{t_2}) \end{array}$$

We assume $unr(x)$, $res(x)$, and $var(x)$ extend to vectors $\vec{x}$ in the expected way. Notice that $var(t) = res(t) \cup unr(t)$ but also that $res(t) \cap unr(t)$ may be non-empty; in $\gamma(\vec{t_1};\vec{t_2})$, terms in $\vec{t_2}$ could contain restricted variables (in nested predicates, for instance).

As hinted at above, the objective of the type system is to identify `lcc` processes whose abstractions contain secure patterns. We consider three kinds of judgments. Judgment $\Delta;\Theta \vdash_{\bullet} c$ concerns patterns: it says that the pattern c is well-formed, under restricted variables Δ and unrestricted variables Θ. The judgment for guards (abstractions, non-deterministic choice) is denoted $\vdash_{\mathbf{A}} G$, whereas a well-typed process P is denoted by $\vdash_{\diamond} P$.

We comment on typing rules for patterns, guards, and processes in Fig. 8. Rules (L:Assoc-l), (L:Assoc-r), and (L:Comm) define basic properties of conjunctions of constraints. Given a predicate $\gamma(\vec{t_1}, \vec{t_2})$, rule (L:Pred) decrees that all variables in $\vec{t_1}$ as well as the variables occurring restricted in $\vec{t_2}$ are restricted. The remaining variables are unrestricted. Rule (L:Comb) identifies the restricted and unrestricted variables in the pattern $c \otimes d$. We require that the set of restricted variables for c must be disjoint from the set of unrestricted variables for d, and viceversa. This avoids treating restricted variables in c or d as unrestricted variables in $c \otimes d$. Typing rules for guards and processes are simple. The most interesting rule is (L:ABS), which says that an abstraction $\forall \vec{x}(d\,;c \to P)$ is secure as long as variables $\vec{x}$ are unrestricted in the typing for c , and no variables in d are abstracted.

The main theorem regarding the type system is *type preservation* (Thm. 5.2), whose proof relies on subject congruence.

Lemma 5.1 (Subject Congruence). *If $P \equiv_l Q$ and $\vdash_\diamond P$, then $\vdash_\diamond Q$.*

Theorem 5.2 (Type Preservation). *If $P \xrightarrow{\alpha} Q$ and $\vdash_\diamond P$ then $\vdash_\diamond Q$.*

Example 5.3 (An Ill-typed Process). As a simple illustration of our type discipline, consider the following process, similar to process $[\![x(y).P]\!]$ in Fig. 6 and to process A discussed in § 5.2:

$$A' = \forall y, w(\mathtt{tt}\,;out(w,y) \otimes \{w{:}x\} \to \overline{in(x,y)} \parallel [\![P]\!])$$

Assume that in $out(y_1, y_2)$ variable y_1 (the endpoint) is restricted and that y_2 (the sent message) is unrestricted; also, suppose that both x_1, x_2 are restricted in $\{x_1{:}x_2\}$. These are natural assumptions: we would like to obtain the message, while protecting communication endpoints from potentially malicious contexts. Using rule (L:COMB), we obtain that pattern $out(w,y) \otimes \{w{:}x\}$ has an unrestricted variable (y) and two restricted variables, w and x. Then, using rule (L:ABS), we infer that process A' is not typable, as it would attempt to perform an insecure abstraction on the restricted variable w.

6. Encoding $\mathsf{s}\pi$ with Session Establishment

We now present our second encoding. We consider $\mathsf{s}\pi^+$, the extension of $\mathsf{s}\pi$ with constructs for *session establishment* based on *explicit locations*. The encoding of $\mathsf{s}\pi^+$ into $\mathtt{lcc}$ builds upon the one given in § 4 to accommodate a *secure* phase of session establishment. We show that our extended encoding maintains the correctness properties of the encoding in § 4 (Cor. 6.8), and is well-typed in the discipline given in § 5 (Thm. 6.9). As such, our extended encoding enjoys a robust treatment of restriction and scope extrusion, ensured by secure patterns.

Next we introduce $\mathsf{s}\pi^+$ (§ 6.1), present its translation into $\mathtt{lcc}$ (§ 6.2), and establish that this translation is an encoding (as in Def. 4.2) and it is well-typed (§ 6.3).

6.1 The Calculus $\mathsf{s}\pi^+$

The syntax of $\mathsf{s}\pi^+$ extends Def. 3.1 with *service requests* and *accepts*, two constructs for representing *session establishment*. Our constructs extend those defined in [16] with information on the *locations* (computation sites) where services reside. Intuitively, two complementary services may establish a session as long as their locations are authorized to do so: a service contains a description of the locations it may interact with. This way, locations are useful to make explicit the fact that services are distributed and that predefined authorization policies govern their interactions.

Formally, let $m, n, \ldots$ range over locations; also, let ρ denote a set of locations. The syntax of $\mathsf{s}\pi^+$ extends $\mathsf{s}\pi$ with two constructs:

- Process $\left[a_y^\rho\langle x\rangle.P\right]^m$ specifies that a declaration (definition) of service a with behavior P resides in location m. Name y denotes an endpoint; both x, y are bound in P. It may only establish sessions with requests from locations included in ρ.
- Process $\left[\overline{a}^m\langle z\rangle.Q\right]^n$ expresses a request of a service named a and located at m. This service request itself resides at n, and has continuation Q. Variable z is bound in Q.

The operational semantics for $\mathsf{s}\pi^+$ extends the reduction relation given in Fig. 1 with the following rule (denoted $\lfloor\text{EST}\rfloor$):

$$\left[\overline{a}^m\langle z\rangle.P\right]^n \mid \left[a_y^\rho\langle x\rangle.Q\right]^m \to_\pi (\boldsymbol{\nu} xy)(P\{y/z\} \mid Q) \quad (n \in \rho)$$

With a slight abuse of notation, we will write $\to_\pi$ to denote a reduction step in $\mathsf{s}\pi^+$. Having constructs for service declaration and request is convenient in specifications. They allow us to describe service names and locations, two elements not present in $\mathsf{s}\pi$. This way, $\mathsf{s}\pi^+$ can be seen as being at a higher abstraction level than $\mathsf{s}\pi$. This convenience is useful for modeling, but it does not represent an expressiveness gain: we can represent service acceptance and request in $\mathsf{s}\pi$. Define the translation $[\![\cdot]\!]^+ : \mathsf{s}\pi^+ \to \mathsf{s}\pi$ as

$$[\![\left[\overline{a}^m\langle z\rangle.P\right]^n]\!]^+ = a_1 \triangleleft m.a_1 \triangleleft n.a_1(z).\,[\![P]\!]^+$$
$$[\![\left[a_y^\rho\langle x\rangle.P\right]^m]\!]^+ = a_2 \triangleright \{m : a_2 \triangleright \{l_i : (\boldsymbol{\nu} xy)(\overline{a_2}\,y \mid [\![P]\!]^+)\}_{l_i \in \rho}\}$$

and as an homomorphism for the other $\mathsf{s}\pi^+$ constructs. Proofs of correctness for this translation exploit the following proposition, which is the key argument for operational correspondence:

Proposition 6.1. *Let $S = \left[\overline{a}^m\langle z\rangle.P\right]^n \mid \left[a_y^\rho\langle x\rangle.Q\right]^m$ be an $\mathsf{s}\pi^+$ process, with $n \in \rho$. Then: If $S \to_\pi S' = (\boldsymbol{\nu} xy)(P\{y/z\} \mid Q)$ then $(\boldsymbol{\nu} a_1a_2)[\![S]\!]^+ \to_\pi^4 (\boldsymbol{\nu} a_1a_2)[\![S']\!]^+$.*

Observe that the encoding $[\![\cdot]\!]^+$ also allows us to reuse the session type system given in § 3.1 for $\mathsf{s}\pi^+$ processes.

6.2 Translating $\mathsf{s}\pi^+$ into $\mathtt{lcc}$

We now present a translation of $\mathsf{s}\pi^+$ into $\mathtt{lcc}$. Key novelties with respect to the encoding given in § 4 are: first, we consider the session establishment phase with locations (as just illustrated, not present in $\mathsf{s}\pi$); second, to ensure that this phase is done correctly, the translation of session declarations/requests implements a simple authentication protocol, the well-known Needham-Schroeder-Lowe (NSL) protocol [20]. To ensure proper authentication with secure patterns, we use the *security* constraint system defined in [15].

6.2.1 A Constraint System for Secure Sessions

In the presence of generalized abstractions with local information (§ 5.1), processes may query the store about local and global constraints. It is crucial to avoid publishing local (restricted) information (e.g., session identifiers, encryption keys, nonces) into the global store. To this end, our translation of $\mathsf{s}\pi^+$ into $\mathtt{lcc}$ relies on a *security* constraint system that combines local and global information with basic cryptographic primitives. Following similar constraint systems in [15, 24, 25], we provide the following definition.

Definition 6.2 (Security Constraint System). *Consider the tuple $\langle \mathcal{C}, \Sigma, \Vdash_C\rangle$ where $\mathcal{C}$ is the set of all constraints obtained by using linear operators $!, \otimes$ and $\exists$ over the function symbols of Σ (Fig. 9), predicate $\mathtt{o}(x)$, and where $\Vdash_C$ is given by the usual deduction rules for linear logic with syntactic equality and the rules in Fig. 10.*

We briefly comment on the signature Σ given in Fig. 9, which differs from that in Fig. 5 in several respects. Function out takes two arguments: a (restricted) session key and an unrestricted message. Function in models the acknowledgment of out, and contains the session key and the value (both restricted). Similarly as before, function sel encodes label selection as a constraint. It contains (a restricted) session key, and the (unrestricted) selected label. Function br models the acknowledgment of sel, and contains the session key and the label (both restricted) that was selected.

The unary predicate $\mathtt{o}(x)$ stands for the output of message x in some global channel. That is, $\mathtt{o}(x)$ says that x is available in some public medium (say, an unprotected network). Function $enc(x; y)$ returns the *encrypted* message y using a key x. Functions $\mathtt{p}(x)$, $\mathtt{r}(x)$, and $\mathtt{s}(x)$ return the public, restricted (private), or symmetric

$$\Sigma \stackrel{\mathtt{def}}{=} in(x,y;\epsilon) \mid out(x;y) \mid sel(x;l) \mid br(x,l;\epsilon) \mid enc(x;y)$$
$$\mid \; covar(x,y;\epsilon) \mid tup_n(\vec{x}) \mid loc_\rho(x) \mid \mathtt{p}(x) \mid \mathtt{r}(x) \mid \mathtt{s}(x)$$

Figure 9. Security constraint system: Function symbols.

$$\text{(E:OUTM)}\ \frac{c \Vdash_C \mathsf{o}(x) \quad c \Vdash_C \mathsf{o}(\gamma(x;m)) \quad \gamma \in \{out, in, sel, br\}}{c \Vdash_C \mathsf{o}(m)}$$

$$\text{(E:COV)}\ \frac{c \Vdash_C \mathsf{o}(x) \quad c \Vdash_C \{x{:}y\} \quad x \neq y}{c \Vdash_C \mathsf{o}(y)}$$

$$\text{(E:KEY)}\ \frac{c \Vdash_C \mathsf{o}(x) \quad k \in \{\mathtt{s},\mathtt{p},\mathtt{r}\}}{c \Vdash_C \mathsf{o}(k(x))} \qquad \text{(E:ENC)}\ \frac{c \Vdash_C \mathsf{o}(x) \quad c \Vdash_C \mathsf{o}(y)}{c \Vdash_C \mathsf{o}(enc(x;y))}$$

$$\text{(E:DEC)}\ \frac{\begin{array}{c} c \Vdash_C \mathsf{o}(k^{-1}(x)) \quad c \Vdash_C \mathsf{o}(enc(k(x);m)) \\ k \in \{\mathtt{s},\mathtt{p}\} \quad \mathtt{s}^{-1} = \mathtt{s}, \mathtt{p}^{-1} = \mathtt{r} \end{array}}{c \Vdash_C \mathsf{o}(m)}$$

$$\text{(E:TUP)}\ \frac{\forall j \in \{1,\ldots,n\}.\, c \Vdash_C \mathsf{o}(i_j)}{c \Vdash_C \mathsf{o}(tup_n(i_1,\ldots,i_n))}$$

$$\text{(E:PROJ)}\ \frac{c \Vdash_C \mathsf{o}(tup_n(i_1,\ldots,i_n)) \quad j \in \{1,\ldots,n\}}{c \Vdash_C \mathsf{o}(i_j)}$$

Figure 10. Security constraint system: Entailment relation.

keys of a channel x, respectively. Function tup_n allow us to create n-ary tuples (with $|\vec{x}| \geq 1$). Function $loc_\rho(x)$ receives a variable x (unrestricted) and a set ρ and returns $\mathtt{1}$ if $x \in \rho$ and $\mathtt{0}$ otherwise.

We comment on the rules of Fig. 10. Rule (E:OUTM) is used to infer session-based communication: given a session key x and a message m with key x (e.g., $\mathsf{o}(out(x;m))$), it is possible to read the message m. Rule (E:COV) relates the two communication endpoints, known only to the participants of that interaction: it states that given an endpoint key x and the co-variable constraint, we may to obtain the key for the other endpoint y. Rule (E:KEY) gives the key of a message. Keys can be public, symmetric or private. Rule (E:ENC) allows us to encode a message x with a given key y. Rule (E:DEC) express that the output of any function of known output values can be inferred using the right key. Rule (E:TUP) allows us to create an n-tuple from a sequence of n-messages. Rule (E:PROJ) defines its destructor, which allows us to project individual elements.

The following notation will be useful in processes.

Notation 6.3. Predicate $enc(x;y)$ will be written as $\{y\}_x$. Also, tuple $tup_n(x_1,\ldots,x_n)$, with $n \geq 1$, will be written $\langle x_1,\ldots,x_n\rangle$.

6.2.2 Translating $\mathsf{s}\pi^+$ into $\mathtt{lcc}$

We now introduce a translation of $\mathsf{s}\pi^+$ with *secure session establishment* into $\mathtt{lcc}$. One of the challenges associated to a translation of session establishment is that the use of abstractions over constraints containing only unrestricted information enables any external process, possibly malicious, to interfere and abstract (guess) session keys, which is clearly undesirable. To solve this issue and guarantee secrecy for the session keys, our translation of session establishment includes an explicit authentication protocol. As mentioned above, we consider the NSL protocol [20]. This choice is orthogonal to the translation; other, more sophisticated protocols could be considered in the translation, which is defined as follows.

$$\begin{aligned}
\llbracket [\overline{a}^m\langle x\rangle.P]^n \rrbracket^{\mathsf{s}}_f &= \exists w.\, \overline{(\{\langle w,n\rangle\}_{\mathtt{p}(m)})} \parallel \\
&\qquad \forall x\big(\mathsf{o}(\mathtt{r}(n))\,;\mathsf{o}(x) \to \overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket P\rrbracket^{\mathsf{s}}_f\big) \\
\llbracket [a^\rho_y\langle x\rangle.P]^m \rrbracket^{\mathsf{s}}_f &= \exists x,y.\big(\forall z,n\big(\mathsf{o}(\mathtt{r}(m))\,;\mathsf{o}(z) \otimes loc_\rho(n) \to \\
&\qquad \overline{\mathsf{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})} \parallel \forall\epsilon(\mathsf{o}(\mathtt{r}(m))\,; \\
&\qquad \mathsf{o}(\{y\}_{\mathtt{p}(m)}) \to \overline{\{x{:}y\}} \parallel \llbracket P\rrbracket^{\mathsf{s}}_f\,)\big)\big) \\
\llbracket \overline{x}\, v.P \rrbracket^{\mathsf{s}}_f &= \overline{\mathsf{o}(out(x,v))} \parallel \\
&\qquad \forall\epsilon(\mathsf{o}(x) \otimes \{x{:}f_x\}\,; in(f_x,v) \to \llbracket P\rrbracket^{\mathsf{s}}_f) \\
\llbracket x(y).P \rrbracket^{\mathsf{s}}_f &= \forall y\big(\mathsf{o}(x) \otimes \{f_x{:}x\}\,; (out(f_x,y)) \to \\
&\qquad \overline{\mathsf{o}(in(x,y))} \parallel \llbracket P\rrbracket^{\mathsf{s}}_f\big) \\
\llbracket x \triangleleft l_i.P \rrbracket^{\mathsf{s}}_f &= \overline{\mathsf{o}(sel(x,l))} \parallel \\
&\qquad \forall\epsilon(\mathsf{o}(x) \otimes \{x{:}f_x\}\,; br(f_x,l) \to \llbracket P\rrbracket^{\mathsf{s}}_f) \\
\llbracket x \triangleright \{l_i{:}P_i\}_{i\in I} \rrbracket^{\mathsf{s}}_f &= \forall l\big(\mathsf{o}(x) \otimes \{f_x{:}x\}\,; sel(f_x,l) \to \\
&\qquad \overline{\mathsf{o}(br(x,l)} \parallel \prod_{1\leq i\leq n} l = l_i \to \llbracket P_i\rrbracket^{\mathsf{s}}_f)\big) \\
\llbracket v?\,(P)\,\mathbf{:}\,(Q) \rrbracket^{\mathsf{s}}_f &= \forall\epsilon(v = \mathtt{tt} \to \llbracket P\rrbracket^{\mathsf{s}}_f\,) \parallel \forall\epsilon(v = \mathtt{ff} \to \llbracket Q\rrbracket^{\mathsf{s}}_f) \\
\llbracket P \mid Q \rrbracket^{\mathsf{s}}_f &= \llbracket P\rrbracket^{\mathsf{s}}_f \parallel \llbracket Q\rrbracket^{\mathsf{s}}_f \qquad \llbracket * \, x(y).P\rrbracket^{\mathsf{s}}_f = !\, \llbracket x(y).P\rrbracket^{\mathsf{s}}_f \\
\llbracket (\boldsymbol{\nu} xy)P \rrbracket^{\mathsf{s}}_f &= \exists x,y.\,(!\,\overline{\{x{:}y\}} \parallel \llbracket P\rrbracket^{\mathsf{s}}_{f\cup\{x:y\}}) \qquad \llbracket \mathbf{0}\rrbracket^{\mathsf{s}}_f = \overline{\mathtt{1}}
\end{aligned}$$

In $\llbracket [\overline{a}^m\langle x\rangle.P]^n \rrbracket^{\mathsf{s}}_f$ we assume $w \notin fv(P)$.
In $\llbracket [a^\rho_y\langle x\rangle.P]^m \rrbracket^{\mathsf{s}}_f$ we assume $z,n \notin fv(P)$.

Figure 11. Translation from $\mathsf{s}\pi^+$ to $\mathtt{lcc}$.

Definition 6.4 ($\mathsf{s}\pi^+$ **into** $\mathtt{lcc}$)**.** *We define the translation from $\mathsf{s}\pi^+$ into $\mathtt{lcc}$ as the tuple $\langle \llbracket\cdot\rrbracket^{\mathsf{s}}_f, \varphi_{\llbracket\cdot\rrbracket^{\mathsf{s}}_f}\rangle$, where:*

(a) $\llbracket\cdot\rrbracket^{\mathsf{s}}_f$ is the process mapping defined in Fig. 11.
(b) $\varphi_{\llbracket\cdot\rrbracket^{\mathsf{s}}_f}$ is defined as in Def. 4.5(b) .

With respect to the encoding of $\mathsf{s}\pi$ into $\mathtt{lcc}$ (cf. Fig. 6), the translation in Fig. 11 follows a similar rationale. Two differences concern the authentication protocol and local information. First, we define a declarative implementation of the NSL protocol. This protocol initiates with the sending of a message w and a location n, encrypted using the public key of the location where the service resides (m). The requested service then creates the two endpoints (noted x, y) and receives a tuple with w and n. Notice that such a tuple is not published publicly, but rather inferred by the possession of the requester's public key and the rules in the constraint system. After that, using its private key, the requested service decodes the tuple message, and encodes the message w, the endpoint y, and location m using with the public key of n. Lastly, the requester receives, decodes, sends back y, encoded with the public key of m: this is to acknowledge that it has received the endpoint. Upon reception, the requested service declares that x and y are co-variables.

Second, the translation in Fig. 11 uses abstractions with local information and secure patterns. Within session communications, constraints of the form $\{x{:}y\}$ (denoting co-variables) are now treated as a pieces of local information, therefore preventing interferences. Generated after session establishment, these constraints are collected in a set (noted f), a parameter of the translation. This is made explicit in the encoding of process $(\boldsymbol{\nu} xy)P$. In Fig. 11, we write f_x to denote the co-variable of x as recorded in f. For convenience, we assume that if $\{x{:}y\} \in f$ then $f_x = y$ and $f_y = x$.

6.3 Correctness of the Translation

We now state correctness of the translation $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f} : \mathtt{s}\pi^{+} \rightarrow \mathtt{lcc}$, in the sense of Def. 4.2. We mostly build upon the definitions and approach given in § 4.3 and § 4.4. The notion of evaluation context is as in Def. 4.7. Besides correctness, we establish typability of encoded processes (Thm. 6.9).

Theorem 6.5 (Name invariance for $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}$). *Let P, σ, and x be a typable $\mathtt{s}\pi^{+}$ process, a substitution satisfying the renaming policy for $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}$ (Def. 6.4(b)), and a variable in* `lcc`*, resp. Then $\llbracket P\sigma\rrbracket^{\mathsf{s}}_{f} = \llbracket P\rrbracket^{\mathsf{s}}_{f}\sigma'$, where $\varphi_{\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}}(\sigma(x)) = \sigma'(\varphi_{\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}}(x))$ and $\sigma = \sigma'$.*

We may also show that our translation is compositional with respect to restriction and parallel.

Theorem 6.6 (Compositionality of $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}$). *Let P and $E[\cdot]$ be a typable $\mathtt{s}\pi^{+}$ process and an $\mathtt{s}\pi^{+}$ evaluation context (cf. Def. 4.7), respectively. Then we have: $\llbracket E[P]\rrbracket^{\mathsf{s}}_{f} = \llbracket E\rrbracket^{\mathsf{s}}_{f}\left[\llbracket P\rrbracket^{\mathsf{s}}_{f}\right]$*

We now state *operational correspondence*. Recall that notation $S \in \wr\llbracket P\rrbracket^{\mathsf{s}}_{f}\int$ has been introduced in Def. 4.11.

Theorem 6.7 (Operational Correspondence for $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}$). *Let P, Q be typable $\mathtt{s}\pi^{+}$ programs and R, S be* `lcc` *processes. Then:*

1. **Soundness:** *If $P \rightarrow_{\pi} Q$ then either:*
   - *(a) $\llbracket P\rrbracket^{\mathsf{s}}_{f} \rightarrow_{l} R$, for some R such that $R \approx \llbracket Q\rrbracket^{\mathsf{s}}_{f}$.*
   - *(b) (or) $\llbracket P\rrbracket^{\mathsf{s}}_{f} \equiv_{l} S' \rightarrow^{2}_{l} R' \equiv_{l} R$, for some R', S', R such that $R \approx \llbracket Q\rrbracket^{\mathsf{s}}_{f}$*
   - *(c) (or) $\llbracket P\rrbracket^{\mathsf{s}}_{f} \rightarrow^{3}_{l} R$, for some R such that $R = \llbracket Q\rrbracket^{\mathsf{s}}_{f}$.*
2. **Completeness:** *If $\llbracket P\rrbracket^{\mathsf{s}}_{f} \rightarrow_{l} S$. Then either:*
   - *(a) $P \rightarrow_{\pi} Q$ for some Q and $\llbracket Q\rrbracket^{\mathsf{s}}_{f} \approx S$.*
   - *(b) (or) $S \in \wr\llbracket P\rrbracket^{\mathsf{s}}_{f}\int$ and $S \rightarrow_{l} S'$, for some S'.
Also, $P \rightarrow_{\pi} Q$ and $\llbracket Q\rrbracket^{\mathsf{s}}_{f} \approx S'$, for some Q.*
   - *(c) (or) $S \in \wr\llbracket P\rrbracket^{\mathsf{s}}_{f}\int$ and $S \rightarrow^{2}_{l} S'$, for some S'.
Also, $P \rightarrow_{\pi} Q$ and $\llbracket Q\rrbracket^{\mathsf{s}}_{f} = S'$, for some Q.*

We discuss some differences with respect to the case of $\mathtt{s}\pi$. The above theorem adds a new possibility for both soundness and completeness (cases (c)). This case takes into account reduction(s) due to session establishment. Since the NSL protocol is a 3-step protocol, three reductions in `lcc` are needed to mimic it. In general, adding a session establishment phase does not affect the operational correspondence results between $\mathtt{s}\pi^{+}$ and `lcc`. In proofs, we reuse most of the definitions required for proving Thm. 4.12. Still, we need to revise the definition of continuation processes (Def. 4.10), in order to consider the new intermediate processes present in session establishment (i.e., session request). Precisely, we extend Def. 4.10 with the following case: if $P \equiv_{\pi} [\overline{a_1}^{m}\langle z\rangle.P]^{n}$, then we have that $⦇P⦈ = \forall x; \mathtt{o}(\mathtt{r}(n)), \mathtt{o}(x) \rightarrow \overline{\mathtt{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket P\rrbracket^{\mathsf{s}}_{f})$. The definition of intermediate process (Def. 4.11) is the same.

Based on the above theorems, we may state:

Corollary 6.8. *Translation $\langle\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}, \varphi_{\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}}\rangle$ is an encoding (cf. Def. 4.2).*

Our final correctness property for the translation is *typability* with respect to the type system in § 5.

Theorem 6.9 (Typability of $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}$). *Let P be a typable $\mathtt{s}\pi^{+}$ process. Then $\vdash_{\diamond} \llbracket P\rrbracket^{\mathsf{s}}_{f}$.*

Proof. By induction on the structure of P. Fig. 12 gives the derivation tree for the case $P = [a^{o}_{y}\langle x\rangle.Q]^{m}$. □

This theorem attests that, provided a disciplined used of patterns (following the signature in Fig. 9), our encoding adheres to a robust interpretation of restriction and scope extrusion. By using secure patterns in our encoding $\llbracket\cdot\rrbracket^{\mathsf{s}}_{f}$, we effectively limit the power of linear abstractions with local information, so as to avoid careless or malicious information leaks related to non-abstractable variables. Indeed, the combination of Theorem 6.9 with Theorems 5.2 and 6.7 (type preservation and operational correspondence, respectively) formalizes static and dynamic robustness guarantees for our declarative representations of structured communications.

7. Related Work

The present developments build upon our previous works [15, 19]. However, because of the substantial technical differences (notably, the presence of linearity) our main results cannot be derived from those in [15, 19]. In particular, differences with respect to [19] include the `ccp` language considered (`lcc` here, `utcc` in [19]). This is a fundamental difference because, as already discussed, thanks to the linear abstractions in `lcc`, our two encodings of $\mathtt{s}\pi$, presented in § 4 and § 6, are rather compact and count with tight operational correspondences. We also improve expressiveness: since `utcc` is a deterministic calculus, the encoding in [19] is unable to capture non-deterministic behavior (as useful in session establishment). In contrast, exploiting linearity, our encoding may capture non-deterministic session establishment (but also forms of non-determinism derivable using unrestricted types in $\mathtt{s}\pi$). Fig. 7 gives a process encodable in our approach but not in [19].

We have shown that the linearity of `lcc` naturally matches the linear communication in $\mathtt{s}\pi$. In `utcc` abstractions are persistent, and so the encoding in [19] is more involved and its operational correspondence is delicate to establish. Intuitively, representing *linear* input prefixes with *persistent* abstractions causes difficulties at several levels. Neither the anomaly of abstraction-based interpretations of scope extrusion/restriction or the use of typing system for secure abstractions to limit abstraction expressivity are addressed in [19]. The type system in [15] and the one in § 5 are similar in spirit, but not in details: the language in [15] is `utcc`, and moving to `lcc` and considering linearity requires non-trivial modifications.

Haemmerlé [13] gives an `lcc` encoding of an asynchronous π-calculus, and establishes operational correspondence for it. His encoding concerns two asynchronous models, which enables a direct operational correspondence. Monjaraz and Mariño [22] encode the asynchronous π-calculus into Flat Guarded Horn Clauses (Flat GHC). They consider compositionality and operational correspondence issues, as we do here. In contrast to [13, 22], here we consider a session π-calculus with synchronous communication, which adds challenges in the encoding and its associated correctness proofs.

The precise relationship between linear logic and session types has been recently established. Caires and Pfenning defined a Curry-Howard interpretation of intuitionistc linear logic as session types [4]. Wadler developed this interpretation for classical linear logic [28]. Giunti and Vasconcelos gave a linear reconstruction of session types [11]; their system is further developed in [27].

Loosely related to our work are [2, 5]. Bocchi et al. [2] integrate declarative requirements into *multiparty* session types by enriching communication descriptions with *logical assertions* which are globally specified within multiparty protocols and potentially projected onto specifications for local participants. Also in the context of choreographies, Carbone et al. [5] explore reasoning via a variant of Hennessy-Milner logic for global and local specifications.

8. Concluding Remarks

We presented two encodings of session π-calculi into `lcc`, a declarative process model based on the `ccp` paradigm. The encodings crucially exploit linearity, an essential common trait in both models. Linearity enables us to define intuitive translations of session-based processes and to state their correctness properties (notably, operational correspondence), improving our previous work [19].

$$
\dfrac{
\dfrac{
\dfrac{}{\vdash_\diamond \overline{\mathtt{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})}}
\qquad
\dfrac{
\dfrac{
\dfrac{}{\vdash_\diamond \overline{\{x{:}y\}}}
\quad
\dfrac{}{\vdash_\diamond [\![Q]\!]^{\mathtt{S}}_f}\;\text{IH}
}{\vdash_\diamond \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_f}
\quad
\dfrac{}{\{y\};\{m\} \vdash_\bullet \mathtt{o}(\{y\}_{\mathtt{p}(m)})}\;(\text{L:}\textsc{Pre})
}{\vdash_\diamond \forall\epsilon(\mathtt{o}(\mathtt{r}(m))\,;\mathtt{o}(\{y\}_{\mathtt{p}(m)}) \rightarrow \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_f\,)}\;(\text{L:}\textsc{Abs})
}{\vdash_\diamond \overline{\mathtt{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})} \parallel \forall\epsilon(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(\{y\}_{\mathtt{p}(m)})) \rightarrow \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_f\,)}\;(\text{L:}\textsc{Par})
\quad
\dfrac{
\dfrac{}{\emptyset;\{z\} \vdash_\bullet \mathtt{o}(z)}\;(\text{L:}\textsc{Pre})
\dfrac{}{\emptyset;\{n\} \vdash_\bullet loc_\rho(n)}\;(\text{L:}\textsc{Pre})
}{\emptyset;\{z,n\} \vdash_\bullet \mathtt{o}(z) \otimes loc_\rho(n)}\;(\text{L:}\textsc{Comb})
}{\vdash_{\mathbf{A}} \forall z,n\big(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(z) \otimes loc_\rho(n) \rightarrow \overline{\mathtt{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})} \parallel \forall\epsilon(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(\{y\}_{\mathtt{p}(m)})) \rightarrow \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_f\,))}\;(\text{L:}\textsc{Abs})
$$

$$
\dfrac{
\dfrac{\vdash_{\mathbf{A}} \forall z,n\big(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(z) \otimes loc_\rho(n) \rightarrow \overline{\mathtt{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})} \parallel \forall\epsilon(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(\{y\}_{\mathtt{p}(m)})) \rightarrow \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_f\,))}{\vdash_\diamond \forall z,n\big(\mathtt{o}(\mathtt{r}(m))\,;((\mathtt{o}(z) \otimes loc_\rho(n) \rightarrow \overline{\mathtt{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})} \parallel \forall\epsilon(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(\{y\}_{\mathtt{p}(m)})) \rightarrow \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_s\,))}\;(\text{L:}\textsc{Guard})
}{\vdash_\diamond \exists x,y.\big(\forall z,n\big(\mathtt{o}(\mathtt{r}(m))\,;((\mathtt{o}(z) \otimes loc_\rho(n) \rightarrow \overline{\mathtt{o}(\{\langle z,y,m\rangle\}_{\mathtt{p}(n)})} \parallel \forall\epsilon(\mathtt{o}(\mathtt{r}(m))\,;(\mathtt{o}(\{y\}_{\mathtt{p}(m)})) \rightarrow \overline{\{x{:}y\}} \parallel [\![Q]\!]^{\mathtt{S}}_f\,)))}\;(\text{L:}\textsc{Loc})
$$

Figure 12. Proof of Theorem 6.9: Typing derivation for $[\![[a^\rho_y\langle x\rangle.Q]^m]\!]^{\mathtt{S}}$.

Our first encoding concerns $\mathsf{s}\pi$, the session π-calculus in [27]; the second encoding considers $\mathsf{s}\pi^+$, an extension of $\mathsf{s}\pi$ with constructs for session establishment. In both cases, we address the correctness of syntactic translations via an abstract notion of encoding, following [12]. The first encoding is representative of our approach, here used for well-studied source and target process languages; the second encoding embodies significant improvements, most notably by considering abstractions with local information (which generalize `lcc` abstractions), explicit authentication protocols for secure session establishment, and a type system that enforces secure abstractions, thus addressing an anomaly of known abstraction-based representations of scope extrusion in the π-calculus.

In future work, we wish to explore whether reasoning techniques for `lcc` processes can support the analysis of $\mathsf{s}\pi$ processes, for instance, to ensure liveness properties (e.g., deadlock-freedom). Also, in order to deepen on the integration of operational and declarative approaches, we plan to extend our encodings to consider the session π-calculus with asynchronous (queue-based), eventful semantics defined in [17].

Acknowledgments. We are grateful to Rémy Haemmerlé, Carlos Olarte, and Vasco Vasconcelos for useful exchanges. We also thank the anonymous reviewers for their valuable suggestions. This research was partially supported by COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems.

References

[1] G. Bernardi, O. Dardha, S. J. Gay, and D. Kouzapas. On duality relations for session types. In *Proc. of TGC'14*, vol. 8902 of *LNCS*, pp. 51–66. Springer, 2014.

[2] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, vol. 6269 of *LNCS*, pp. 162–176. Springer - Verlag, 2010.

[3] M. G. Buscemi and U. Montanari. Cc-pi: A constraint language for service negotiation and composition. In *Results of the SENSORIA Project*, vol. 6582 of *LNCS*, pp. 262–281. Springer, 2011.

[4] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010*, LNCS, pp. 222–236. Springer, 2010.

[5] M. Carbone, D. Grohmann, T. T. Hildebrandt, and H. A. López. A logic for choreographies. In *Proc. of PLACES 2010*, vol. 69 of *EPTCS*, pp. 29–43, 2010.

[6] M. Coppo and M. Dezani-Ciancaglini. Structured communications with concurrent constraints. In *Proc. of TGC 2008*, vol. 5474 of *LNCS*, pp. 104–125. Springer, 2009.

[7] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.

[8] J. F. Díaz, C. Rueda, and F. D. Valencia. Pi+- calculus: A calculus for concurrent processes with constraints. *CLEI Electron. J.*, 1(2), 1998.

[9] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput.*, 165(1):14 – 41, 2001.

[10] J.-Y. Girard. Linear logic. *Theor. Comp. Sci.*, 50:1–102, 1987.

[11] M. Giunti and V. T. Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR*, LNCS, pp. 432–446. Springer, 2010.

[12] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[13] R. Haemmerlé. Observational equivalences for linear logic concurrent constraint languages. *TPLP*, 11(4-5):469–485, 2011.

[14] R. Haemmerlé and H. Betz. Verification of constraint handling rules using linear logic phase semantics. In *The 5th Workshop on Constraint Handling Rules*, no. 08 –10 in RISC-Linz Report Series, pp. 67–78, 2008.

[15] T. Hildebrandt and H. A. López. Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming . In *Proc. of ICLP 2009*, vol. 5649 of *LNCS*, pp. 417–431. Springer, 2009.

[16] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of ESOP'98*, vol. 1381, pp. 122–138. Springer, 1998.

[17] D. Kouzapas, N. Yoshida, and K. Honda. On asynchronous session semantics. In *Proc. of FORTE'11*, vol. 6722 of *LNCS*, pp. 228–243. Springer, 2011.

[18] C. Laneve and U. Montanari. Mobility in the cc-paradigm. In *Proc. of MFCS'92*, vol. 629 of *LNCS*, pp. 336–345. Springer, 1992.

[19] H. A. López, C. Olarte, and J. A. Pérez. Towards a Unified Framework for Declarative Structured Communications. In *Proc. of PLACES 2009*, vol. 17 of *EPTCS*, pp. 1–15, 2010.

[20] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Software - Concepts&Tools*, 17(3):93–102, 1996.

[21] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[22] R. Monjaraz and J. Mariño. From the π-calculus to flat GHC. In *Proc. of PPDP'12*, pp. 163–172. ACM, 2012.

[23] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic J. of Computing*, 9(1):145–188, 2002.

[24] C. Olarte and F. D. Valencia. The expressivity of universal timed CCP: undecidability of monadic FLTL and closure operators for security. In *Proc. of PPDP'08*, pp. 8–19. ACM, 2008.

[25] C. A. Olarte and F. D. Valencia. Universal concurrent constraint programming: Symbolic semantics and applications to security. In *SAC'08*, pp. 145–150, New York, NY, USA, 2008. ACM.

[26] V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[27] V. T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

[28] P. Wadler. Propositions as sessions. In *Proc. of ICFP'12*, pp. 273–286. ACM, 2012.

$$\text{(T:Bool)}\ \frac{un(\Gamma)}{\Gamma \vdash \mathtt{ff}, \mathtt{tt} : q\,bool} \qquad \text{(T:Var)}\ \frac{un(\Gamma_1, \Gamma_2)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T} \qquad \text{(T:Nil)}\ \frac{un(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

$$\text{(T:Par)}\ \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{(T:Res)}\ \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P}$$

$$\text{(T:In)}\ \frac{\Gamma_1 \vdash x : q\,?T.U \quad (\Gamma_2, y : T) \circ x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}$$

$$\text{(T:Out)}\ \frac{\Gamma_1 \vdash x : q\,!T.U \quad \Gamma_2 \vdash v : T \quad \Gamma_3 \circ x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}\,v.P}$$

$$\text{(T:Sel)}\ \frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \quad \Gamma_2 \circ x : T_j \vdash P \quad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P}$$

$$\text{(T:Bra)}\ \frac{\Gamma_1 \vdash x : q\ \&\ \{l_i : T_i\}_{i \in I} \quad \forall i \in I.\, \Gamma_2 \circ x : T_i \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i \in I}}$$

$$\text{(T:If)}\ \frac{\Gamma_1 \vdash v : q\,bool \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash v?\,(P)\mathbf{:}(Q)} \qquad \text{(T:Repl)}\ \frac{\Gamma \vdash P \quad un(\Gamma)}{\Gamma \vdash *\,P}$$

Figure 13. Full typing rules for $\mathtt{s}\pi$ processes.

A. Omitted Definitions

A.1 Context Splittling

Definition A.1 (Context splitting). *Let* Γ_1, Γ_2 *denote concatenation of contexts* Γ_1 *and* Γ_2*. Context splitting is defined as follows:*

$$\emptyset \circ \emptyset = \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad un(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad lin(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad lin(T)}{\Gamma, x : T = \Gamma_1 \circ (\Gamma_2, x : T)}$$

A.2 Complete Set of Typing Rules for $\mathtt{s}\pi$

The complete set of rules is presented in Fig. 13. We give an intuition on the rules of the typing system not described in the main text: rules (T:BOOL) and (T:VAR) are for variables; in both cases, we check that all linear variables are consumed, using predicate $un(\cdot)$. Rule (T:NIL) types the inactive process **0**; it also checks that the context only contains unrestricted variables. Rule (T:IF) type-checks the conditional process. Rule (T:REPL) checks replicated processes, making sure that the associated context is unrestricted.

B. Appendix to Section 4

We present proofs for Theorem 4.12 in Page 7.

Theorem 4.12 (Operational Correspondence). Statement on page 7.

Proof. We detail the proofs of soundness (1) and completeness (2) separately:

1. **Soundness:** The proof is by induction on the reduction for $\mathtt{s}\pi$.
   Case Rule $\lfloor\text{IFT}\rfloor$:
   (i) Assume $P = \mathtt{tt}?\,(P')\mathbf{:}(P'')$
   (ii) By (i) then $P \rightarrow_\pi P'$ using $\lfloor\text{IFT}\rfloor$.
   (iii) Then by the application of the definition of $\llbracket\cdot\rrbracket$ (Def. 4.5):

$$\begin{aligned}\llbracket P\rrbracket =& \forall\epsilon(\mathtt{tt} = \mathtt{tt} \rightarrow \llbracket P'\rrbracket) \parallel \\ & \forall\epsilon(\mathtt{tt} = \mathtt{ff} \rightarrow \llbracket P''\rrbracket)\end{aligned}$$

(iv) By using rule (C:SYNC) (Fig. 4), with $c = 1$ we have that (note that $\Vdash \texttt{tt} = \texttt{tt}$)

$$\llbracket P \rrbracket \rightarrow_l \llbracket P' \rrbracket \parallel \forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow \llbracket P'' \rrbracket)$$

(v) By (iv) note that the process

$$\forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow \llbracket P'' \rrbracket)$$

is blocked, the constraint $\texttt{tt} = \texttt{ff}$ cannot be satisfied. Then, conclude that $\llbracket P' \rrbracket_l \approx \llbracket P' \rrbracket_l \parallel \forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow \llbracket P'' \rrbracket)$.

Case $\lfloor$IFF$\rfloor$: Analogous to previous case.

Case $\lfloor$COM$\rfloor$:

(i) Assume $P = (\boldsymbol{\nu} xy)(\overline{x}\, v.P' \mid y(z).P'')$

(ii) By (i) $P \rightarrow_\pi (\boldsymbol{\nu} xy)(P' \mid P''\{{}^{v}\!/z\})$ using $\lfloor$COM$\rfloor$.

(iii) By definition of $\llbracket \cdot \rrbracket$:

$$\begin{aligned}\llbracket P \rrbracket = \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \overline{(out(x, v)} \parallel \\ \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ \forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow \\ (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket))\end{aligned}$$

(iv) By using the rules of structural congruence and reduction of `lcc` one can build the following reduction:

$$\begin{aligned}\llbracket P \rrbracket \equiv_l\ & \exists x, y.\, ((\overline{!\{x{:}y\} \otimes out(x, v)} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow \\ & (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket)) \\ \rightarrow_l\ & \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \overline{in(y, v)} \parallel \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket) \\ \equiv_l\ & \exists x, y.\, ((\overline{!\{x{:}y\} \otimes in(y, v)} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket \\ \rightarrow_l\ & \exists x, y.((\overline{!\{x{:}y\}} \parallel \llbracket P'\{{}^{y}\!/z\} \rrbracket \parallel \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket)\end{aligned}$$

(v) Conclude by considering the form of the process obtained in the previous derivation as follows:

$$\begin{aligned}&\llbracket (\boldsymbol{\nu} xy)(P' \parallel P''\{{}^{v}\!/z\}) \rrbracket = \\ &\exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \llbracket P'\{{}^{y}\!/z\} \rrbracket \parallel \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket)\end{aligned}$$

Case Rule $\lfloor$REPL$\rfloor$: Analogous to case $\lfloor$COM$\rfloor$, as follows:

(i) Assume $P = (\boldsymbol{\nu} xy)(\overline{x}\, v.P' \mid *\, y(z).P'')$

(ii) By (i) $P \rightarrow_\pi (\boldsymbol{\nu} xy)(P' \mid P''\{{}^{v}\!/z\} \mid *\, y(z).P'')$ using $\lfloor$REP$\rfloor$.

(iii) By definition of $\llbracket \cdot \rrbracket$:

$$\begin{aligned}\llbracket P \rrbracket =\ & \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \overline{(out(x, v)} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & !\forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow \\ & (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket)) \\ \equiv_l\ & \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \overline{(out(x, v)} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow \\ & (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket)) \parallel \\ & !\forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow \\ & (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket))\end{aligned}$$

(iv) Let

$$R = !\forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket))$$

Then, by using the rules of structural congruence and reduction of `lcc` one can build the following reduction:

$$\begin{aligned}\llbracket P \rrbracket \equiv_l\ & \exists x, y.\, ((\overline{!\{x{:}y\} \otimes out(x, v)} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \forall z, w(out(w, z) \otimes \{w{:}y\}) \rightarrow \\ & (\overline{in(y, z)} \parallel \llbracket P'' \rrbracket)) \parallel R \\ \rightarrow_l\ & \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \overline{in(y, v)} \parallel \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket) \parallel R \\ \equiv_l\ & \exists x, y.\, ((\overline{!\{x{:}y\} \otimes in(y, v)} \parallel \\ & \forall z((in(z, v) \otimes \{x{:}z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\ & \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket \parallel R \\ \rightarrow_l\ & \exists x, y.((\overline{!\{x{:}y\}} \parallel \llbracket P'\{{}^{y}\!/z\} \rrbracket \parallel \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket \\ & \parallel R)\end{aligned}$$

(v) Conclude by considering the form of the process obtained in the previous derivation as follows:

$$\begin{aligned}&\llbracket (\boldsymbol{\nu} xy)(P' \mid P''\{{}^{v}\!/z\} \mid *\, y(z).P'') \rrbracket = \\ &\exists x, y.\, (\overline{!\{x{:}y\}} \parallel \llbracket P'\{{}^{y}\!/z\} \rrbracket \parallel \llbracket P''\{{}^{v,x}\!/z, w\} \rrbracket \parallel R)\end{aligned}$$

Case Rule $\lfloor$SEL$\rfloor$: Analogous to input case.

Completeness: Directly by appealing to the structure of an encoded well-formed, typable program (Not. 3.5). Without loss of generality, we can reduce the proof the the minimal number of processes that can interact: a single conditional process and a single pair of interacting processes (i.e., input/output, branching/selection, output/replication).

(i) By considering the encoded form of a well-formed program (Lem. 4.9) we have:

$$\llbracket P \rrbracket \equiv_l \exists \vec{x}, \vec{y}.\, (\llbracket R_1 \rrbracket \parallel \cdots \parallel \llbracket R_n \rrbracket \parallel V)$$

Where each $R_i, 0 \leq i \leq n$ is a pre-redex.

(ii) From the hypothesis, if $P \not\rightarrow_l 1$. Vacuously true.

(iii) From the (i), we know that each R_i is a pre-redex.

(iv) By the application of (i,ii,iii) note that if $P \rightarrow_l$ then, there is at least a pre-redex (or pair of pre-redexes) that can reduce. Without loss of generality, consider the cases where there is only 1 and 2 pre-redexes and they can reduce:

Case $R_i = v?\,(R_i')\,\mathbf{:}\,(R_i'')$ (there exists only one pre-redex):

(i) If R_i can reduce, then $v = \texttt{tt} \vee v = \texttt{ff}$, thus we distinguish two cases:

- **Subcase** $v = \texttt{tt}$:

  (i) We show that $R_i \rightarrow_\pi R_i'$ by using rule $\lfloor$IFT$\rfloor$.

  (ii) Conclude by setting $Q = R_i'$ (Thus showing such Q exists). Note that

  $$\llbracket R_i' \rrbracket \parallel \forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow \llbracket P'' \rrbracket) \approx \llbracket R_i' \rrbracket$$

- **Subcase** $v = \texttt{ff}$: Analogous to the previous case.

Case There exists two pre-redexes: we consider three cases: input/output, selection/branching, replication/output. We present the case for input/output, the others are analogous.

Subcase Input/output interaction:

(i) Assume $R_i = \overline{x}\, v.R_i'$ and $R_j = y(z).R_j'$

(ii) By (i) and the hypotheses ($P \rightarrow_l$) we set $P = (\boldsymbol{\nu} xy)(\overline{x}\, v.R_i' \parallel R_j = y(z).R_j)$

(iii) We show a reduction similar for that of the case $\lfloor \textsc{Com} \rfloor$ in soundness for $[\![P]\!]$ and show that

$$S \in \wr [\![P]\!] \int \wedge S = \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \forall \epsilon((in(y,v) \otimes \{x{:}y\}) \rightarrow [\![P']\!]) \parallel \overline{in(y,v)} \parallel [\![P''\{^{v}\!/_{z}\}]\!])$$

And that $S \rightarrow_l S'$ such that

$$S' = \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P''\{^{v}\!/_{z}\}]\!])$$

(iv) Set $Q = (\boldsymbol{\nu} xy)(P' \parallel P''\{^{v}\!/_{z}\}])$, and by the reduction rules of $\mathsf{s}\pi$ we have that $P \rightarrow_\pi Q$, and conclude: $[\![Q]\!] \approx S'$

Subcase $R_i = \overline{x}\, v.R'_i$ and $R_j = *\, y(z).R'_j$: Analogous to the input case; the only consideration is the replicated input process, which is treated as in case $\lfloor \textsc{Com} \rfloor$ of the completeness result.

Subcase $R_i = x \triangleleft l_i.R'_i$ and $R_j = x \triangleright \{l_i : P_i\}_{i \in I}$: is similar to the previous cases. We again appeal to bisimilarity, since $S' \approx [\![Q]\!]$, then again because there are blocked "garbage" processes (i.e., processes that will not be able to execute).

□

C. Appendix to Section 5

We give proofs for Subject Congruence (Lemma 5.1) and Type Preservation (Theorem 5.2).

Definition C.1 (Substitutions). *Given terms $\vec{t} = t_1, \ldots, t_n$ and process variables $\vec{x} = x_1, \ldots, x_n$, the application of a substitution to a constraint, guard and process, denoted respectively $c\{^{\vec{t}}\!/_{\vec{x}}\}$, $G\{^{\vec{t}}\!/_{\vec{x}}\}$, and $P\{^{\vec{t}}\!/_{\vec{x}}\}$, is inductively defined on the structure of constraints, guards and process as:*

$$
\begin{array}{ll}
\mathtt{1}\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} \mathtt{1} \\
\mathtt{0}\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} \mathtt{0} \\
\gamma(\vec{u};\vec{v})\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} \gamma(\vec{u}\{^{\vec{t}}\!/_{\vec{x}}\};\vec{v}\{^{\vec{t}}\!/_{\vec{x}}\}) \\
(c \otimes d)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} c\{^{\vec{t}}\!/_{\vec{x}}\} \otimes d\{^{\vec{t}}\!/_{\vec{x}}\} \\
(\exists \vec{y}.c)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} \exists \vec{y}.c\{^{\vec{t}}\!/_{\vec{x}}\} \quad (\vec{y} \cap \vec{x} = \emptyset) \\
(!c)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} !(c\{^{\vec{t}}\!/_{\vec{x}}\}) \\
\forall \vec{y}(d\,;c \rightarrow P)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} \forall \vec{y}(d\{^{\vec{t}}\!/_{\vec{x}}\}\,; c\{^{\vec{t}}\!/_{\vec{x}}\} \rightarrow P\{^{\vec{t}}\!/_{\vec{x}}\}) \quad (\vec{y} \cap \vec{x} = \emptyset) \\
(G_1 + G_2)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} G_1\{^{\vec{t}}\!/_{\vec{x}}\} + G_2\{^{\vec{t}}\!/_{\vec{x}}\} \\
(\overline{c})\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} \overline{c\{^{\vec{t}}\!/_{\vec{x}}\}} \\
(P \parallel Q)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} P\{^{\vec{t}}\!/_{\vec{x}}\} \parallel Q\{^{\vec{t}}\!/_{\vec{x}}\} \\
(\exists \vec{y}.\, P) & \stackrel{\text{def}}{=} \exists \vec{y}.\, P\{^{\vec{t}}\!/_{\vec{x}}\} \quad (\vec{y} \cap \vec{x} = \emptyset) \\
(!\, P)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} !\,(P\{^{\vec{t}}\!/_{\vec{x}}\}) \\
(G)\{^{\vec{t}}\!/_{\vec{x}}\} & \stackrel{\text{def}}{=} G\{^{\vec{t}}\!/_{\vec{x}}\}
\end{array}
$$

Lemma 5.1 (Subject Congruence). Statement on page 9.

Proof. The proof proceeds by induction on the depth of the premise $P \equiv_l Q$.

Case

rule premise	$P \parallel \overline{1} \equiv_l P$	(1)
rule premise	$\vdash_\diamond P \parallel \overline{1}$	(2)
2, inversion	$\vdash_\diamond P$	(3)

Case

rule premise	$\exists z.\, \overline{1} \equiv_l \overline{1}$	(1)
rule premise	$\vdash_\diamond \exists z.\, \overline{1}$	(2)
2, inversion	$\vdash_\diamond \overline{1}$	(3)

Case

rule premise	$\exists x.\, \exists y.\, P \equiv_l \exists y.\, \exists x.\, P$	(1)
rule premise	$\vdash_\diamond \exists x.\, \exists y.\, P$	(2)
2, inversion	$\vdash_\diamond \exists y.\, P$	(3)
3, inversion	$\vdash_\diamond P$	(4)
4, formation	$\vdash_\diamond \exists x.\, P$	(5)
5, formation	$\vdash_\diamond \exists y.\, \exists x.\, P$	(6)

Case

rule premise	$!P \equiv_l P \parallel !P$	(1)
rule premise	$\vdash_\diamond !P$	(2)
2, inversion	$\vdash_\diamond P$	(3)
2, 3, formation	$\vdash_\diamond P \parallel !P$	(4)

Case

rule premise	$\overline{c} \parallel \overline{d} \equiv_l \overline{e}$	(1)
rule premise	$\vdash_\diamond \overline{c} \parallel \overline{d}$	(2)
1, inversion	$c \otimes d \dashv\vdash_C e$	(3)
2, inversion	$\vdash_\diamond \overline{c}$	(4)
2, inversion	$\vdash_\diamond \overline{d}$	(5)
4, inversion	$c \in \mathcal{C}$	(6)
5, inversion	$d \in \mathcal{C}$	(7)
6,7	$c \otimes d \in \mathcal{C}$	(8)
3,8, entailment	$e \in \mathcal{C}$	(9)
9, formation	$\vdash_\diamond \overline{e}$	(10)

Case

rule premise	$P \parallel Q \equiv_l P' \parallel Q$	(1)
rule premise	$\vdash_\diamond P \parallel Q$	(2)
1, inversion	$P \equiv_l P'$	(3)
2, inversion	$\vdash_\diamond P$	(4)
2, inversion	$\vdash_\diamond Q$	(5)
3,4, IH	$\vdash_\diamond P'$	(6)
5,6, formation	$\vdash_\diamond P' \parallel Q$	(7)

Case

rule premise	$P \parallel \exists z.\, Q \equiv_l \exists z.\, (P \parallel Q)$	(1)
rule premise	$\vdash_\diamond P \parallel \exists z.\, Q$	(2)
1, inversion	$z \notin fv(P)$	(3)
2, inversion	$\vdash_\diamond P$	(4)
2, inversion	$\vdash_\diamond \exists z.\, Q$	(5)
5, inversion	$\vdash_\diamond Q$	(6)
4,6, formation	$\vdash_\diamond P \parallel Q$	(7)
7, formation	$\vdash_\diamond \exists z.\, (P \parallel Q)$	(8)

Case

rule premise	$\exists x.\, P \equiv_l \exists x.\, P'$	(1)
rule premise	$\vdash_\diamond \exists x.\, P$	(2)

1, inversion	$P \equiv_l P'$	(3)
2, inversion	$\vdash_\diamond P$	(4)
3, 4, IH	$\vdash_\diamond P'$	(5)
5, formation	$\vdash_\diamond \exists x.\, P'$	(6)

□

Proposition C.2. *Let P and t be a process and a term, respectively. If $\vdash_\diamond P$ then $\vdash_\diamond P\{t/x\}$.*

Proof. By induction on the structure of P. Interesting cases are when $P = \overline{c}$ and $P = \forall\vec{y}(d\,;e \to P')$, for some $\vec{y}$, d, e, and P'; other cases are easy.

- Case $P = \overline{c}$: By the well-typedness assumption we infer that $c \in \mathcal{C}$, which immediately implies that $c\{t/x\} \in \mathcal{C}$ and so we conclude using (L:TELL).
- Case $P = \forall\vec{y}(d\,;e \to P')$: By the well-typedness assumption we infer that P' is well-typed, that $\Delta;\Theta \vdash_\bullet e$, and that $\vec{y} \subseteq dom(\Theta) \setminus fv(d)$. Since universal and existential quantifiers are binders, occurrences of variables in $\vec{y}$ are not affected by $\{t/x\}$; this in particular rules out the possibility of renaming an unrestricted variable into a restricted one. The substitution thus only affects free variables (not in $\vec{y}$) and the thesis follows.

□

Theorem 5.2 (Type Preservation). Statement on page 9.

Proof. The proof proceeds by induction on the depth of the premise $P \xrightarrow{\alpha} Q$.

Case (C:OUT)

rule premise	$\overline{c} \xrightarrow{(\vec{x'})\overline{d'}} \overline{e}$	(1)
1, inversion	$c \Vdash_C \exists\vec{x}(d \otimes e)$	(2)
1, inversion	$\exists\vec{x}d \Vdash_C \exists\vec{x'}d'$	(3)
1, inversion	$\{\vec{x}, \vec{x'}\} \cap fv(c) = \emptyset$	(4)
rule premise	$\vdash_\diamond \overline{c}$	(5)
5, inversion	$c \in \mathcal{C}$	(6)
2, 6, transitivity	$e \in \mathcal{C}$	(7)
7, formation	$\vdash_\diamond \overline{e}$	(8)

Case (C:SYNLOC)

premise	$\overline{c} \parallel \forall\vec{x}(d\,;e \to P) \xrightarrow{\tau} \exists\vec{y}.\,(P\{\vec{t}/\vec{x}\} \parallel \overline{f})$	(1)
1, inversion	$c \otimes d \Vdash_C \exists\vec{y}.(e\{\vec{t}/\vec{x}\} \otimes f)$	(2)
1, inversion	$\vec{y} \cap fv(c, d, e, P) = \emptyset$	(3)
1, inversion	$\mathbf{mgc}\big(c \otimes d, \exists\vec{y}.(e\{\vec{t}/\vec{x}\} \otimes f)\big)$	(4)
1, inversion	$c \otimes d \Vdash_C \mathsf{0} \Rightarrow c \Vdash_C \mathsf{0}$	(5)
hypothesis	$\vdash_\diamond \overline{c} \parallel \forall\vec{x}(d\,;e \to P)$	(6)
6, inversion	$\vdash_\diamond \overline{c}$	(7)
6, inversion	$\vdash_\diamond \forall\vec{x}(d\,;e \to P)$	(8)
8, inversion	$\vdash_{\mathbf{A}} \forall\vec{x}(d\,;e \to P)$	(9)
9, inversion	$\vdash_\diamond P$	(10)
9, inversion	$\Delta;\Theta \vdash_\bullet e$	(11)
9, inversion	$\vec{x} \subseteq dom(\Theta) \setminus fv(d)$	(12)
2, inversion	$\exists\vec{y}.(e\{\vec{t}/\vec{x}\} \otimes f) \in \mathcal{C}$	(13)
13, transitivity	$f \in \mathcal{C}$	(14)
14, formation	$\vdash_\diamond \overline{f}$	(15)
4, def. $\mathbf{mgc}$	$\vec{t}$ is a vector of terms	(16)
10, 16, Prop. C.2	$\vdash_\diamond P\{\vec{t}/\vec{x}\}$	(17)
15, 17, form.	$\vdash_\diamond P\{\vec{t}/\vec{x}\} \parallel \overline{f}$	(18)
18, formation	$\vdash_\diamond \exists\vec{y}.\,(P\{\vec{t}/\vec{x}\} \parallel \overline{f})$	(19)

Case (C:IN)

premise	$\overline{\mathtt{1}} \xrightarrow{c} \overline{c}$	(1)
premise	$\vdash_\diamond \overline{\mathtt{1}}$	(2)
2, inversion	$\mathtt{1} \in \mathcal{C}$	(3)
1, well-formedness of labels	$c \in \mathcal{C}$	(4)
4, formation	$\vdash_\diamond \overline{c}$	(5)

Case (C:COMP)

premise	$P \parallel Q \xrightarrow{\alpha} P' \parallel Q$	(1)
1, inversion	$P \xrightarrow{\alpha} P'$	(2)
1, inversion	$ev(\alpha) \cap fv(Q) = \emptyset$	(3)
premise	$\vdash_\diamond P' \parallel Q$	(4)
4, inversion	$\vdash_\diamond P$	(5)
4, inversion	$\vdash_\diamond Q$	(6)
2, 5, IH	$\vdash_\diamond P'$	(7)
6, 7, formation	$\vdash_\diamond P' \parallel Q$	(8)

Case (C:SUM) (Shown for $i = 1$, the other case is identical)

premise	$P \parallel G_1 + G_2 \xrightarrow{\alpha} Q$	(1)
1, inversion	$P \parallel G_1 \xrightarrow{\alpha} Q$	(2)
premise	$\vdash_\diamond P \parallel G_1 + G_2$	(3)
3, inversion	$\vdash_\diamond P$	(4)
3, inversion	$\vdash_\diamond G_1 + G_2$	(5)
5, inversion	$\vdash_{\mathbf{A}} G_1 + G_2$	(6)
6, inversion	$\vdash_{\mathbf{A}} G_1$	(7)
7, formation	$\vdash_\diamond G_1$	(8)
4, 8, formation	$\vdash_\diamond P \parallel G_1$	(9)
2,9, IH	$\vdash_\diamond Q$	(10)

Case (C:EXT)

premise	$\exists y.\, P \xrightarrow{(yx)\overline{c}} Q$	(1)
premise	$\vdash_\diamond \exists y.\, P$	(2)
1, inversion	$P \xrightarrow{(x)\overline{c}} Q$	(3)
2, inversion	$\vdash_\diamond P$	(4)
3,4, IH	$\vdash_\diamond Q$	(5)

Case (C:CONG)

premise	$P_1 \xrightarrow{\alpha} P_2$	(1)
premise	$\vdash_\diamond P_1$	(2)
1, inversion	$P_1 \equiv_l P_1'$	(3)
1, inversion	$P_1' \xrightarrow{\alpha} P_2'$	(4)
1, inversion	$P_2' \equiv_l P_2$	(5)
2, 3, lemma 5.1	$\vdash_\diamond P_1'$	(6)
4,6, IH	$\vdash_\diamond P_2'$	(7)
5,7, lemma 5.1	$\vdash_\diamond P_2$	(8)

Case (C:RES)

premise	$\exists y.\, P \xrightarrow{\alpha} \exists y.\, Q$	(1)
1, inversion	$P \xrightarrow{\alpha} Q$	(2)
1, inversion	$y \notin fv(\alpha)$	(3)
premise	$\vdash_\diamond \exists y.\, P$	(4)
4, inversion	$\vdash_\diamond P$	(5)
2,5, IH	$\vdash_\diamond Q$	(6)
6, formation	$\vdash_\diamond \exists y.\, Q$	(7)

□

D. Appendix to Section 6

Theorem 6.7 (Operational Correspondence for $\llbracket\cdot\rrbracket^{\mathtt{S}}_f$). Statement on page 11.

Proof. We detail the proofs of soundness (1) and completeness (2) separately:

1. **Soundness:**
   - The proof maintains the structure of the previous operational correspondence. The only new case is the session establishment rule case:
     (i) The new rule is:
     $$([\overline{a}^{l_2}\langle x\rangle.P]^{l_1} \mid [a^{\rho}_{y}\langle z\rangle.Q)]^{l_2} \rightarrow_\pi (\boldsymbol{\nu} zy)(P\{y/x\} \mid Q)$$
     Where $l_1 \in \rho$
     (ii) By the definition of the encoding:
     $$\begin{aligned}
     &\llbracket([\overline{a}^{l_2}\langle x\rangle.P]^{l_1} \mid [a^{\rho}_{y}\langle z\rangle.Q)]^{l_2}\rrbracket^{\mathtt{S}}_f\\
     &= \exists m.\, (\overline{\mathsf{o}(\{\langle m, l_1\rangle\}_{\mathtt{p}(l_2)})} \parallel\\
     &\quad \forall x(\mathsf{o}(\mathtt{r}(l_1))\,;\mathsf{o}(x) \rightarrow \overline{\mathsf{o}(\{x\}_{\mathtt{p}(l_2)})} \parallel \llbracket P\rrbracket^{\mathtt{S}}_f)) \parallel\\
     &\quad \exists z, y.\, (\forall w, l_3(\mathsf{o}(\mathtt{r}(l_2))\,;(\mathsf{o}(w) \otimes loc_\rho(l_3)) \rightarrow\\
     &\qquad (\overline{\mathsf{o}(\{\langle w, y, l_2\rangle\}_{\mathtt{p}(l_3)})} \parallel\\
     &\qquad \forall\epsilon(\mathsf{o}(\mathtt{r}(l_2))\,;(\mathsf{o}(\{y\}_{\mathtt{p}(l_2)})) \rightarrow \overline{!\{z{:}y\}} \parallel \llbracket Q\rrbracket^{\mathtt{S}}_f))))\\
     &\xrightarrow{\tau} \exists z, y, m.\, (\forall x(\mathsf{o}(\mathtt{r}(l_1))\,;\mathsf{o}(x) \rightarrow (\overline{\mathsf{o}(\{x\}_{\mathtt{p}(l_2)})}) \parallel \llbracket P\rrbracket^{\mathtt{S}}_f) \parallel\\
     &\qquad (\overline{\mathsf{o}(\{\langle m, y, l_2\rangle\}_{\mathtt{p}(l_1)})} \parallel\\
     &\qquad \forall\epsilon(\mathsf{o}(\mathtt{r}(l_2))\,;(\mathsf{o}(\{y\}_{\mathtt{p}(l_2)})) \rightarrow \overline{!\{z{:}y\}} \parallel \llbracket Q\rrbracket^{\mathtt{S}}_f)))\\
     &\xrightarrow{\tau} \exists z, y, m.\, ((\overline{\mathsf{o}(\{y\}_{\mathtt{p}(l_2)})} \parallel \llbracket P\{y/x\}\rrbracket^{\mathtt{S}}_f) \parallel\\
     &\qquad \forall\epsilon(\mathsf{o}(\mathtt{r}(l_2))\,;(\mathsf{o}(\{y\}_{\mathtt{p}(l_2)})) \rightarrow \overline{!\{z{:}y\}} \parallel \llbracket Q\rrbracket^{\mathtt{S}}_f)))\\
     &\xrightarrow{\tau} \exists z, y, m.\, (\llbracket P\{y/x\}\rrbracket^{\mathtt{S}}_f \parallel \overline{!\{z{:}y\}} \parallel \llbracket Q\rrbracket^{\mathtt{S}}_f)
     \end{aligned}$$
     (iii) Since m is a new variable that does not exists in P or Q, conclude:
     $$\begin{aligned}
     &\exists z, y, m.\, (\llbracket P\{y/x\}\rrbracket^{\mathtt{S}}_f \parallel \overline{!\{z{:}y\}} \parallel \llbracket Q\rrbracket^{\mathtt{S}}_f)\\
     &\equiv_l \exists z, y.\, (\llbracket P\{y/x\}\rrbracket^{\mathtt{S}}_f \parallel \overline{!\{z{:}y\}} \parallel \llbracket Q\rrbracket^{\mathtt{S}}_f)\\
     &= \llbracket(\boldsymbol{\nu} zy)(P\{y/x\} \mid Q)\rrbracket^{\mathtt{S}}_f
     \end{aligned}$$
   - For the sake of illustration, we show the new structure for rule $\lfloor$COM$\rfloor$, note that $\{x{:}y\} \in f$; thus $f_x = y, f_y = x$. All the other cases proceed similarly
     (i) Assume $P = (\boldsymbol{\nu} xy)(\overline{x}\, v.P' \mid y(z).P'')$
     (ii) By (i) $P \rightarrow_\pi (\boldsymbol{\nu} xy)(P' \mid P''\{v/z\})$ using $\lfloor$COM$\rfloor$.
     (iii) By definition of $\llbracket\cdot\rrbracket^{\mathtt{S}}_f$:
     $$\begin{aligned}
     \llbracket P\rrbracket^{\mathtt{S}}_f = \exists x, y.\, ((&\overline{!\{x{:}y\}} \parallel \overline{\mathsf{o}(out(x, v))} \parallel\\
     &\forall\epsilon(\mathsf{o}(x) \otimes \{x{:}f_x\}\,; in(f_x, v) \rightarrow \llbracket P'\rrbracket^{\mathtt{S}}_f) \parallel\\
     &\forall z(\mathsf{o}(y) \otimes \{f_y{:}y\}\,; out(f_y, z) \rightarrow\\
     &\qquad (\overline{\mathsf{o}(in(y, z))} \parallel \llbracket P''\rrbracket^{\mathtt{S}}_f))
     \end{aligned}$$
     (iv) By using the rules of structural congruence and reduction of `lcc` one can build the following reduction:
     $$\begin{aligned}
     \llbracket P\rrbracket^{\mathtt{S}}_f &\equiv_l \exists x, y.\, ((\overline{!\{x{:}y\} \otimes \mathsf{o}(out(x, v))} \parallel\\
     &\qquad \forall\epsilon(\mathsf{o}(x) \otimes \{x{:}f_x\}\,; in(f_x, v) \rightarrow \llbracket P'\rrbracket^{\mathtt{S}}_f) \parallel\\
     &\qquad \forall z(\mathsf{o}(y) \otimes \{f_y{:}y\}\,; out(f_y, z) \rightarrow\\
     &\qquad\qquad (\overline{\mathsf{o}(in(y, z))} \parallel \llbracket P''\rrbracket^{\mathtt{S}}_f))\\
     &\xrightarrow{\tau} \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel\\
     &\qquad \forall\epsilon(\mathsf{o}(x) \otimes \{x{:}f_x\}\,; in(f_x, v) \rightarrow \llbracket P'\rrbracket^{\mathtt{S}}_f) \parallel\\
     &\qquad \overline{\mathsf{o}(in(y, v))} \parallel \llbracket P''\{v/z\}\rrbracket^{\mathtt{S}}_f)\\
     &\equiv_l \exists x, y.\, ((\overline{!\{x{:}y\} \otimes \mathsf{o}(in(y, v))} \parallel\\
     &\qquad \forall\epsilon(\mathsf{o}(x) \otimes \{x{:}f_x\}\,; in(f_x, v) \rightarrow \llbracket P'\rrbracket^{\mathtt{S}}_f) \parallel\\
     &\qquad \llbracket P''\{v/z\}\rrbracket^{\mathtt{S}}_f\\
     &\xrightarrow{\tau} \exists x, y.((\overline{!\{x{:}y\}} \parallel \llbracket P'\rrbracket^{\mathtt{S}}_f \parallel \llbracket P''\{v/z\}\rrbracket^{\mathtt{S}}_f)
     \end{aligned}$$
     (v) Conclude by considering the form of the process obtained in the previous derivation as follows:
     $$\begin{aligned}
     &\llbracket(\boldsymbol{\nu} xy)(P' \parallel P''\{v/z\})\rrbracket^{\mathtt{S}}_f =\\
     &\exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \llbracket P'\rrbracket^{\mathtt{S}}_f \parallel \llbracket P''\{v/z\}\rrbracket^{\mathtt{S}}_f)
     \end{aligned}$$
2. **Completeness:** The proof has the same structure as the proof of Theorem 4.12. The new case is given for the case where there are two pre-redexes interacting (session establishment pre-redexes) and it is analogous to the case for input/output. The case corresponds to the following interaction:
   $$\begin{aligned}
   P &= \llbracket([\overline{a}^{l_2}\langle x\rangle.P]^{l_1} \mid [a^{\rho}_{y}\langle z\rangle.Q)]^{l_2}\rrbracket^{\mathtt{S}}_f\\
   &= \exists m.\, (\overline{\mathsf{o}(\{\langle m, l_1\rangle\}_{\mathtt{p}(l_2)})} \parallel\\
   &\quad \forall x(\mathsf{o}(\mathtt{r}(l_1))\,;\mathsf{o}(x) \rightarrow \overline{\mathsf{o}(\{x\}_{\mathtt{p}(l_2)})} \parallel \llbracket P\rrbracket^{\mathtt{S}}_f)) \parallel\\
   &\quad \exists z, y.\, (\overline{!\{z{:}y\}} \parallel\\
   &\quad \forall w, l_3(\mathsf{o}(\mathtt{r}(l_2))\,;(\mathsf{o}(w) \otimes l_3 \in \rho) \rightarrow\\
   &\qquad (\overline{\mathsf{o}(\{\langle w, y, l_2\rangle\}_{\mathtt{p}(l_3)})} \parallel\\
   &\qquad \forall\epsilon(\mathsf{o}(\mathtt{r}(l_2))\,;(\mathsf{o}(\{y\}_{\mathtt{p}(l_2)})) \rightarrow \llbracket Q\rrbracket^{\mathtt{S}}_f))))
   \end{aligned}$$

   It is possible to see (by building a similar reduction for the case of soundness) that in one reduction we will have process $S \in \wr\llbracket P\rrbracket^{\mathtt{S}}_f\wr$, and that in 2 reductions and one structural congruence step we reach $\llbracket Q\rrbracket^{\mathtt{S}}_f$ as desired.

□

Theorem 6.9 (Well-typedness for $\llbracket\cdot\rrbracket^{\mathtt{S}}_f$). Statement on page 11.

Proof. The proof proceeds by induction on the structure of P.

Case $P = [\overline{a}^{m}\langle x\rangle.Q]^{n}$: The derivation tree is given in Fig. 14.
Case $P = [a^{\rho}_{y}\langle x\rangle.Q]^{m}$: The derivation tree is given in Fig. 12.
Case $P = \overline{x}\, v.Q$: The derivation tree is given in Fig. 15.
Case $P = x(y).Q$: The derivation tree is given in Fig. 16.
Case $P = x \triangleleft l_i.P$: The derivation tree is given in Fig. 17.
Case $P = x \triangleright \{l_i : Q_i\}_{i\in I}$: The derivation tree is given in Fig. 18. Note that when using the inductive hypothesis, we previously need to use n steps with (L:PAR).
Case $P = v?\,(Q) : (R)$: The proof is trivial as all the variables of an equality are unrestricted; we omit the derivation tree.

$$\dfrac{\dfrac{}{\vdash_\diamond \overline{\mathsf{o}(\{\langle w,n\rangle\}_{\mathtt{p}(m)})}}\;\text{(L:\textsc{Tell})} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_\diamond \llbracket Q\rrbracket^{\mathsf{S}}_f}\;\text{IH} \quad \dfrac{}{\vdash_\diamond \overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})}}\;\text{(L:\textsc{Tell})}}{\vdash_\diamond \overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f} \qquad \dfrac{}{\emptyset;\{x\} \vdash_\bullet \mathsf{o}(x)}\;\text{(L:\textsc{Pred})}}{\vdash_{\mathbf{A}} \forall x\big(\mathsf{o}(\mathtt{r}(n))\,;\mathsf{o}(x)\to\overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f\big)}}{\vdash_\diamond \forall x\big(\mathsf{o}(\mathtt{r}(n))\,;\mathsf{o}(x)\to\overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f\big)}\;\text{(L:\textsc{Guard})}}{\dfrac{\vdash_\diamond \big(\overline{\mathsf{o}(\{\langle w,n\rangle\}_{\mathtt{p}(m)})} \parallel \forall x\big(\mathsf{o}(\mathtt{r}(n))\,;\mathsf{o}(x)\to\overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f\big)}{\vdash_\diamond \exists w.\,\big(\overline{\mathsf{o}(\{\langle w,n\rangle\}_{\mathtt{p}(m)})} \parallel \forall x\big(\mathsf{o}(\mathtt{r}(n))\,;\mathsf{o}(x)\to\overline{\mathsf{o}(\{x\}_{\mathtt{p}(m)})} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f\big)}\;\text{(L:\textsc{Local})}}\;\text{(L:\textsc{Par})}$$

Figure 14. Typing derivation for $\llbracket[\overline{a}^m\langle x\rangle.Q]^n\rrbracket^{\mathsf{S}}_f$.

$$\dfrac{\dfrac{}{\vdash_\diamond \overline{\mathsf{o}(out(x;v))}}\;\text{(L:\textsc{Tell})} \qquad \dfrac{\dfrac{\dfrac{}{\vdash_\diamond \llbracket Q\rrbracket^{\mathsf{S}}_f}\;\text{IH} \qquad \dfrac{}{\{f_x,v\};\emptyset \vdash_\bullet in(f_x,v;\epsilon)}\;\text{(L:\textsc{Pred})}}{\vdash_{\mathbf{A}} \forall\epsilon(\mathsf{o}(x)\otimes\{x{:}f_x\}\,;in(f_x,v;\epsilon)\to\llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Abs})}}{\vdash_\diamond \forall\epsilon(\mathsf{o}(x)\otimes\{x{:}f_x\}\,;in(f_x,v;\epsilon)\to\llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Guard})}}{\vdash_\diamond \overline{\mathsf{o}(out(x;v))} \parallel \forall\epsilon(\mathsf{o}(x)\otimes\{x{:}f_x\}\,;in(f_x,v;\epsilon)\to\llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Par})}$$

Figure 15. Typing derivation for $\llbracket\overline{x}\,v.Q\rrbracket^{\mathsf{S}}_f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_\diamond (\overline{\mathsf{o}(in(x,y;\epsilon))}}\;\text{(L:\textsc{Tell})} \qquad \dfrac{}{\vdash_\diamond \llbracket Q\rrbracket^{\mathsf{S}}_f}\;\text{IH}}{\vdash_\diamond (\overline{\mathsf{o}(in(x,y;\epsilon))} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Par})} \qquad \dfrac{}{\{f_x\};\{y\} \vdash_\bullet out(f_x;y)}\;\text{(L:\textsc{Pred})}}{\vdash_{\mathbf{A}} \forall y\big(\mathsf{o}(x)\otimes\{f_x{:}x\}\,;(out(f_x;y))\to\overline{\mathsf{o}(in(x,y;\epsilon))} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f\big)}\;\text{(L:\textsc{Abs})}}{\vdash_\diamond \forall y\big(\mathsf{o}(x)\otimes\{f_x{:}x\}\,;(out(f_x;y))\to\overline{\mathsf{o}(in(x,y;\epsilon))} \parallel \llbracket Q\rrbracket^{\mathsf{S}}_f\big)}\;\text{(L:\textsc{Guard})}$$

Figure 16. Typing derivation for $\llbracket x(y).Q\rrbracket^{\mathsf{S}}_f$.

$$\dfrac{\dfrac{}{\vdash_\diamond \overline{\mathsf{o}(sel(x;l))}}\;\text{(L:\textsc{Tell})} \qquad \dfrac{\dfrac{\dfrac{}{\vdash_\diamond \llbracket Q\rrbracket^{\mathsf{S}}_f}\;\text{IH} \qquad \dfrac{}{\{f_x,l\};\emptyset \vdash_\bullet br(f_x,l;\epsilon)}\;\text{(L:\textsc{Pred})}}{\vdash_{\mathbf{A}} \forall\epsilon(\mathsf{o}(x)\otimes\{x{:}f_x\}\,;br(f_x,l;\epsilon)\to\llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Abs})}}{\vdash_\diamond \forall\epsilon(\mathsf{o}(x)\otimes\{x{:}f_x\}\,;br(f_x,l;\epsilon)\to\llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Guard})}}{\vdash_\diamond \overline{\mathsf{o}(sel(x;l))} \parallel \forall\epsilon(\mathsf{o}(x)\otimes\{x{:}f_x\}\,;br(f_x,l;\epsilon)\to\llbracket Q\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Par})}$$

Figure 17. Typing derivation for $\llbracket x \triangleleft l.Q\rrbracket^{\mathsf{S}}_f$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_\diamond (\overline{\mathsf{o}(sel(x;l))}}\;\text{(L:\textsc{Tell})} \qquad \dfrac{}{\vdash_\diamond \prod_{1\le i\le n} l=l_i \to \llbracket Q_i\rrbracket^{\mathsf{S}}_f}\;\text{IH}}{\vdash_\diamond (\overline{\mathsf{o}(br(x,l;\epsilon))} \parallel \prod_{1\le i\le n} l=l_i \to \llbracket Q_i\rrbracket^{\mathsf{S}}_f)}\;\text{(L:\textsc{Par})} \qquad \dfrac{}{\{f_x\};\{l\} \vdash_\bullet sel(f_x;l)}\;\text{(L:\textsc{Pred})}}{\vdash_{\mathbf{A}} \forall l\big(\mathsf{o}(x)\otimes\{f_x{:}x\}\,;sel(f_x;l)\to\overline{\mathsf{o}(br(x,l;\epsilon)} \parallel \prod_{1\le i\le n} l=l_i\to\llbracket Q_i\rrbracket^{\mathsf{S}}_f)\big)}\;\text{(L:\textsc{Abs})}}{\vdash_\diamond \forall l\big(\mathsf{o}(x)\otimes\{f_x{:}x\}\,;sel(f_x;l)\to\overline{\mathsf{o}(br(x,l;\epsilon)} \parallel \prod_{1\le i\le n} l=l_i\to\llbracket Q_i\rrbracket^{\mathsf{S}}_f)\big)}\;\text{(L:\textsc{Guard})}$$

Figure 18. Typing derivation for $\llbracket x \triangleright \{l_i : Q_i\}_{i\in I}\rrbracket^{\mathsf{S}}_f$.

□