Declarative Interpretations of Session-Based Concurrency

Mauricio Cano and Camilo Rueda
Pontificia Universidad Javeriana - Cali

Hugo A. López
Technical University of Denmark

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, expressiveness.

1. Introduction

This paper relates two distinct models of concurrent processes: one of them, the session π-calculus (sπ), is inherently operational; the other one, given by linear concurrent constraint programming (lcc), 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 against which specifications may be checked. As these specifications can be conveniently given in the π-calculus, 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 models describe how a communicating system is implemented, declarative models 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., $\llbracket\llbracket\llbracket$. On the other hand, declarative models of concurrency naturally express partial and contextual information (see, e.g., $\llbracket\llbracket\llbracket\llbracket$. Although some of these models may represent forms of π-calculus mobility, 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 $\llbracket\llbracket$ 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 $\llbracket\llbracket$, as declarative processes in universal concurrent constraint programming (utcc). 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 (encodings) of sπ processes as declarative processes in linear concurrent constraint programming (lcc). Our interpretations follow the approach in $\llbracket\llbracket\llbracket\llbracket$), 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 and to the verification of safety properties for sπ processes. As for (b), we address this anomaly by extending lcc with abstractions with local information and by developing a simple type system for lcc processes. Building upon $\llbracket\llbracket\llbracket\llbracket$, 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 sπ.

Next, we illustrate further our approach and contributions. $\llbracket\llbracket\llbracket\llbracket$ introduces sπ and lcc. A first interpretation of sπ in lcc is given in $\llbracket\llbracket$. The type system for lcc is presented in $\llbracket\llbracket\llbracket\llbracket$. Our second interpretation, given in $\llbracket\llbracket\llbracket\llbracket$, considers $\pi^*: the extension of sπ with session establishment. On top of this interpretation, we use types for lcc to ensure correct abstractions. We close by discussing related work ($\llbracket\llbracket\llbracket$) and some concluding remarks ($\llbracket\llbracket\llbracket$. The Appendix contains additional technical details (e.g., proofs).
2. Overview

A Basic Scenario. Consider the following simple protocol between a client and an online store:

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 protocol may be expressed using session types as follows. From the client’s perspective, the session type

\[ T_c = \text{item.?price}. @ \{ \text{buy : } B_c, \text{quit : !bye, end} \} \]

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 (@) between two behaviors denoted by labels buy and quit, and abstracted by types \( B_c \) and \( \text{!bye, end} \), respectively.

To illustrate the relationship between session types and \( \pi \)-calculus processes, we introduce some notation. We write \( \pi(v.P) \) and \( x(y).P \) to denote output and input along name \( x \), with continuation \( P \). Also, we write \( x <\ell\lab \) to represent selection of a label \( \lab \), and \( b?P \) (\( Q \)) 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 \( x \):

\[ P_x = \pi \text{book}. x(z). (z \leq 20)? (x \triangleleft \text{buy, } R_c); (x \triangleleft \text{quit, } \text{!end, 0}) \]

where \( R_c \) implements the purchase routine (described by type \( B_c \)).

The relationship between session types (such as \( T_c \)) and processes in the \( \pi \)-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 \[1,28\].

The Basic Scenario with Declarative Conditions. The session \( \pi \)-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 \( \pi \)-calculus.

We thus argue that the (standard) \( \pi \)-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. In this paper, we focus on process models based on concurrent constraint programming (ccp) \[25\]. 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 \( \pi \)，the session \( \pi \)-calculus in \[27, 17\], and \( 1cc \) \[9, 13\]. We introduce two declarative interpretations (encodings) of \( \pi \)-calculus into \( 1cc \) 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 \( \pi \) can be transposed to the declarative world of \( 1cc \) through our interpretations.

Our Contributions. A common trait in \( \pi \) and \( 1cc \) is linearity: it enables to enforce disciplined resource-aware session protocols; linearity is also central to \( 1cc \), as we explain next.

Let \( c, d \) and \( x \) denote constraints and a (possibly empty) vector of variables. While the tell process \( \tau \) can be seen as the output of \( c \) to the store, the linear abstraction \( \forall x(d \rightarrow P) \) may be intuitively read as: if \( d \) 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 \( x \) in \( P \), cf. \[3,22\]. When \( x \) is empty, we write \( \forall c(d \rightarrow P) \).

Here we develop two interpretations of well-typed \( \pi \)-processes into \( 1cc \). In the first interpretation (denoted \[1,14\]) and given in \[4\], output \( \pi \triangleright v.P \) and input \( x(y).Q \) in \( \pi \) are encoded as

\[
[\pi \triangleright v.P] = [\pi(out(v,x) \triangleright \forall x(in(z,v) \otimes [x \leq 20] \rightarrow [P]))
\]

\[
[x(y).Q] = [\forall x, w(out(w,y) \otimes [w \triangleleft x : info(in(x,y) \triangleright [Q]))
\]

Predicates \( out(v,x) \) and \( in(y,x) \) are used to model synchronous communication in \( \pi \); constraint \( [x \geq 20] \) says that \( x \) and \( z \) are two dual session endpoints. These pieces of information are treated as linear resources by \( 1cc \); this is critical to ensure faithfulness of the interpretation with respect to the source \( \pi \) process (cf. Thm.14,7). This interpretation attests the expressivity of linear abstractions in representing name passing and scope extrusion in \( \pi \).

Using \[3,12\] we can already give \( 1cc \) specifications which combine representations of \( \pi \)-calculus 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 \( \pi \). As an example, consider \( \pi \)-processes \( P_{\text{buy}} = x \triangleleft \text{buy, } R_c \) and \( P_{\text{quit}} = x \triangleleft \text{quit, } \text{!end, 0} \), two sub-processes of \( P_x \) above. In order to represent step 3’ given before, using our interpretation \[1,14\] we could define the \( 1cc \) process

\[
\forall c(z \leq 20) \rightarrow [P_{\text{buy}}] \parallel \forall c(z > 20) \rightarrow [P_{\text{quit}}]
\]

which uses conjunction to add the presence of event \( e \) into the decision of buying the item ([P_{\text{buy}}]) or quitting the protocol ([P_{\text{quit}}]).

It turns out that linear abstractions are overly powerful: they may express forms of scope extrusion not possible in \( \pi \) (see \[3,22\]). To overcome this anomaly, our second interpretation (denoted \[1,17\] and given in \[6\], 1cc encodes an extension of \( \pi \) with session establishment (sriers) using linear abstractions with local information:

\[
\forall x(d; c \rightarrow P)
\]

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.

In \[1,17\], the session key (used by the two end-points) is treated as local information in the encoding of session synchronizations. Let \( o(m) \) denote a predicate representing the output of message \( m \) in a public medium. In the presence of local information, session output \( \pi \triangleright v.P \) and input \( x(y).Q \) are represented roughly as follows:

\[
[\pi \triangleright v.P] = \delta(out(v,x)) \parallel \forall o(x) \triangleright \forall w(in(v,w) \rightarrow [P])
\]

\[
[x(y).Q] = \forall y(o(x) \triangleright \forall w.out(w,y) \rightarrow \delta(in(x,y) \triangleright [Q])
\]

2015/6/18
By treating constraint o(x) ⊗ [x:w] as local information, we prevent
interference from (malicious) participants aiming at intercepting
the identity of session endpoints x and w.

Although helpful, the use of local information does not suffice
to properly limit the expressivity of abstractions: we need to disal-
low abstracting on variables that should remain local to the session.

To this end, in §5 we introduce a type system on lcc processes that
covers variables in abstractions. By distinguishing between rest-
icted and unrestricted variables in predicates, we shall say that an
lcc process P is well-typed if for every abstraction ∀x(x : e → P)
in P, variables x are unrestricted. This guarantees that malicious
attackers cannot infer private information, such as session keys. We
show that our second interpretation enjoys operational correspon-
dence (Thm. 6.7) and is well-typed (Thm. 6.9), in the above sense.

3. Preliminaries

We now introduce the models that we formally work in this level:
the session π-calculus of [27] (§3.1) and lcc lcc [27] (§3.2). Notation
e denotes a sequence of elements e1, e2, . . . , enn, with length |e| = n.

3.1 The session π-calculus (π)

Syntax. Assume a countable infinite set of variables V e, ranged
over by x, y, . . . . For simplicity, we only consider boolean con-
stants (tt, ff); we use v, v′, . . . to range over variables and con-
stants (values). Also, we use l, l′, . . . to range over labels.

Definition 3.1 (π Processes). The syntax for π processes is given
by the following grammar:

\[ P, Q ::= π v.P \mid x(y).P \mid x \ll l.P \mid \nu [l_i : R_i] \mid \nu [\nu v.P] \mid [0] \]

Process π 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 \ll l.P uses x to
select l from a labeled choice process x \ll \{l_i : Q_i\} so as to
trigger Qj. We assume pairwise distinct labels. Process \nu v.P denotes
a replicated input process; it allows one to express infinite behaviors.

Operational Semantics. The semantics for π processes is given
as a reduction relation, denoted \rightarrow e, defined as the smallest rela-
tion generated by the rules in Fig.1. Reduction expresses the com-
putation steps that a process performs on its own. It relies on a

![Figure 1. Reduction relation for π processes.](image)

Figure 1. Reduction relation for π processes.

structural congruence on processes, denoted \equiv, which identifies
processes up to consistent renaming of bound names, denoted \equiv o.

Formally, \equiv o is the smallest congruence that satisfies the axioms:

\[ P \equiv o P \mid Q \equiv o P \mid P \equiv o Q \]

Types are qualified pretypes or recursive types for disciplin-
ing potentially infinite communication patterns. Intuitively, linearly
qualified types are assigned to endpoints occurring in exactly one
thread (a process not comprising parallel composition); the unre-
stricted 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 T o to denote the dual of type T .

Given a context Γ and a process P, typing judgments are of the
form Γ ⊢ 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 o, which maintains the linearity in-
variant 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:BS) types an input
process: it checks whether x has the right type and checks the con-
notation; 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:BR:A) and (T:SE:Is) type-check
branching and selection processes, respectively.

We state the subject reduction property for this type system:
We now collect some results that concern the structure of processes. An arbitrary predicate is denoted \( \gamma \), and it is well-typed under the empty environment (Lemma 3.6 [27]).

Definition 3.7 (lcc syntax). The syntax for \( \text{lcc} \) is given by the following grammar:

\[
\begin{align*}
c &::= 1 | 0 | \gamma(l) | c \odot c | \exists \vec{x}. c | \neg c \\
P &::= \forall \vec{x}(c \rightarrow P) | G + G \\
\tau &::= \parallel Q \parallel P | \parallel P | G \parallel G
\end{align*}
\]

The grammar for constraints \( c \) defines the pieces of information that will be posted (asked) to (from) the store. Constraint 1, the multiplicative identity, denotes truth; constant 0 denotes falsehood. Predicates are denoted \( \gamma(l) \). Formulas are built from the multiplicative conjunction (\( \odot \)), 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 \( \forall \vec{x}(c \rightarrow P) \). Processes \( P \) include guards as well as the tell operator (denoted \( \tau \)) and constructs for parallel composition (\( \parallel \)), hiding (\( \exists \vec{x} \)), and replication (\( ! \)), which have expected readings as forms of concurrency, local and infinite behavior, respectively. Notation \( \prod_{i \leq n} P_i \) (with \( n \geq 1 \)) stands for the process \( P_1 || \cdots || P_n \). Universal and existential quantifiers are variable binders. The free variables of constraints and processes are denoted \( \forall \vec{t}(\vec{y}) \). We write \( c(\vec{y}, \vec{z}) \) to denote the constraint obtained by the (capture-avoiding) substitution of the free variables of \( x_i \) for \( t_i \) in \( c \), with \( [\vec{y}] = [\vec{z}] \) and pairwise distinct \( x_i \)’s. Process substitution \( P(\vec{y}/\vec{x}) \) is defined analogously.

Semantics. We follow the semantics for \( \text{lcc} \) processes given by Haemmerlé [14], which is defined as a Labeled Transition System (LTS) and considers a set of linear constraints \( C \) and an entailment relation \( \models_C \) over \( C \). We notice that \( C \) is parametric on a given set of predicates, and so \( C \) may change according to signature \( \Sigma \). The LTS relies on a structural congruence on processes, given next.

Definition 3.8 (Structural Congruence). The structural congruence relation for \( \text{lcc} \) processes is the smallest congruence relation \( \equiv_1 \) which satisfies the following rules:

\[
\begin{align*}
P || \Gamma \equiv_1 P \equiv_1 \exists x. \Gamma \equiv_1 \exists x. \exists y. P \equiv_1 \exists x. P \equiv_1 \exists y. P \equiv_1 \exists x. P \\
P \parallel Q \equiv_1 P \parallel Q \equiv_1 \forall \vec{x}(c \rightarrow P) \equiv_1 \forall \vec{x}(\vec{y}. P) \equiv_1 \forall \vec{x}. P \\
\exists \vec{x}. Q \equiv_1 \exists \vec{x}. (P || Q) \equiv_1 \exists \vec{x}. P
\end{align*}
\]

A transition \( P \xrightarrow{\alpha} P' \) denotes the evolution of process \( P \) to \( P' \) by performing the action denoted by label \( \alpha \).

\[
\alpha \ ::= \tau | c | (\vec{x}) \tau
\]

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

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

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

\[
mgc(c, \exists \vec{x}.(d(\vec{t}/\vec{x}) \odot e))
\]

whenever for any constraint \( e' \), all terms \( \vec{t} \) and all variables \( \vec{y}' \), if \( c \uplus C \models \exists \vec{x}.(d(\vec{t}/\vec{x}) \odot e) \) and \( \exists \vec{y}'e' \uplus C \models \exists \vec{x}. \exists \vec{y}'. \nu \) hold, then \( \exists \vec{y}'(d(\vec{t}/\vec{x})) \uplus C \models \exists \vec{y}'. \nu \) and \( \exists \vec{y}' \uplus C \models \exists \vec{y}'. \nu \).

Intuitively, the \( mgc \) 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.
(C:OUT) \[ c \parallel c \quad \exists \tilde{d}(d \parallel e) \quad \exists \tilde{d}'d' \quad \exists \tilde{d}' \quad (\exists \tilde{x}(\tilde{d} \parallel e) \cap f(v)(c) = \emptyset) \]

(C:IN) \[ \exists \tilde{x} \quad \exists \tilde{z} \quad \exists \tilde{x}' \quad \exists \tilde{z}' \quad (\exists \tilde{x}(\tilde{d} \parallel e) \cap f(v)(c) = \emptyset) \]

We will assume that mgc, the set of all constraints. The observables of an lcc process P is the set of all D-accessible constraints, as defined in [13].

Definition 3.10 (D-accessible constraints). Let D \subseteq C, where C is the set of all constraints. The observables of an lcc process P is the set of all D-accessible constraints defined as follows:

\[ D(P) = \{ (x,e) \in D \mid \text{there exists } P', P \xrightarrow{\pi} \exists x.(P' \parallel \tau) \} \]

We now define a bisimulation relation for lcc processes. Let D, E \subseteq C. We say that an action (label) \( \rho \) is D-E-relevant for a process P if it is either a silent action \( \tau \) or an input action in E, or an output action in E with \( \exists \tilde{x} \cap f(v)(c) = \emptyset \) and \( \exists \tilde{x} \in D \).

Definition 11 (D-E-bisimulation). Let D, E \subseteq C, where C is the set of all constraints. Then we say that a symmetric relation \( R \) is a D-E-bisimulation if for all processes P, Q and for all labels \( \alpha \) such that PRQ, P \xrightarrow{\alpha} P' and Q is a D-E-relevant action for Q, there exists Q' s.t. Q \xrightarrow{\alpha} Q' and P'RQ'. The largest D-E-bisimulation is called D-E-bisimulation and is denoted \( \approx_{D,E} \).

We will assume D = E = C; bisimilarity will be denoted by \( \approx \).

4. Encoding \( s’t \) in lcc

In this work, our interest is in encodings, i.e., language translations which enjoy certain encodability criteria. This section develops an encoding of \( s’t \) into lcc. We first present the formal notion of encoding that we consider ([3]). Then, we describe the encoding ([4]) and establish its correctness ([5], [6]).

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 \( L \) is a tuple \( (P, \xrightarrow{\cdot}, \approx) \), where \( P \) is a set of processes, \( \xrightarrow{\cdot} \) denotes an operational semantics, and \( \approx \) is a behavioral equality on \( P \).

We give the set of \( \text{rules (C:R)} \) are self-explanatory. Rules (C:EV) and (C:RES) formalize hiding. Finally, rule (C:CONG) closes transitions under structural congruence (cf. Def. [3]).

Weak transitions are standardly defined; we write \( P \xrightarrow{\tau} Q \) iff \( (P \xrightarrow{\tau} Q) \) and \( P \xrightarrow{\tau} Q \) iff \( (P \xrightarrow{\tau} Q) \). Reduction \( P \xrightarrow{\cdot} Q \) is defined as \( P \equiv P' \xrightarrow{\cdot} Q \). We write \( \xrightarrow{\cdot} \) 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 D-accessible constraints, as defined in [13].

Definition 3.10 (D-accessible constraints). Let D \subseteq C, where C is the set of all constraints. The observables of an lcc process P is the set of all D-accessible constraints defined as follows:

\[ O^D(P) = \{ (\exists x.e) \in D \mid \text{there exists } P', P \xrightarrow{\pi} \exists x.(P' \parallel \tau) \} \]

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 \( s’t \) into lcc

We move to consider \( s’t \) and lcc as source and target languages in a translation. We first define the constraint system that we will use:
Omitting some unimportant substitutions, the translation of \( P \) intuitively behaves as follows. Observe how process \( \text{out}(x,v) \), 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 \( \text{in}(y,v) \parallel [P_2] \{ y/u \} \), containing the continuation of the input. Once that occurs, a similar pattern in the reverse direction is performed: constraint \( \text{in}(y,v) \) and the co-variable constraint will trigger the continuation of the output, denoted \([P]_1\). 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(b).

### 4.3 Translation Correctness (1): Static Properties

We show that \([P]\) 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 \([P]\)).** Let \( P, \sigma, \) and \( x \) be a typable \( \pi \) process, a substitution satisfying the renaming policy for \([P]\) (Def. 4.5(b)), and a variable in \( \sigma \), resp. Then \([P\sigma] = [P]\sigma'\), with \( \varphi_1\pi(\sigma(x)) = \sigma'\varphi_1\pi(x) \) and \( \sigma = \sigma' \).

To simplify the presentation of the semantic properties, we define the usual notion of evaluation contexts for \( \pi \).

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

\[
[\pi] ::= \cdot | E | P | P | (\nu xy) \langle E \rangle
\]

An (evaluation) context is a process with a “hole”, denoted \( \cdot \). Given an evaluation context \( [\pi] \), we write \( E[P] \) to denote the \( \pi \) 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., \( (\nu xy) \cdot \).

We now prove compositionality of \( [\pi] \) with respect to restriction and parallel composition operator, as in Def. 4.2(2).

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

Having established static criteria for \([\pi]\), we now investigate operational correspondence, a dynamic encodability criterion.

### 4.4 Translation Correctness (2): Operational Correspondence

One important issue to be addressed with \([\cdot]\) is the non-determinism of \( \pi \). This 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 \( \pi \) program below, not encodable in \[19\]:

\[
Q = (\nu xy)(\pi v_1.P_1 \parallel \pi v_2.P_2 \parallel y(z).R)
\]

which is typable in \[27\] with a context \( \Gamma = \{ x : \mu a.unbool.T, y : in?bool.U \} \). We have either

\[
Q \rightarrow (\nu xy)(\pi v_1.P_1 \parallel \pi v_2.P_2 \parallel R(v_1/z)) = Q_1
\]

\[
Q \rightarrow (\nu xy)(\pi v_1.P_1 \parallel P_2 \parallel R(v_2/z)) = Q_2
\]

Now consider the \( \lcc \) process \([Q]\):

\[
\exists x, y, \{ x:y \} \parallel \text{out}(x,v_1) \parallel \text{in}(z,v_1) \rightarrow [P_1] \parallel \text{out}(x,v_1) \parallel \text{in}(z,v_2) \rightarrow [P_2] \parallel \text{out}(w,z) \parallel \text{in}(y,z) \parallel [R]
\]

One can show that \([Q]\) reaches a state in which only one of the outputs will interact with the input process; Fig. 7 details associated transitions. Given the definition of \([\cdot]\), 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
Figure 7. Evolution of the \(lcc\) translation of program \([1]\) (§4.4). 

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 \([Q]\) may also evolve into \([Q_2]\).

Observe the process obtained before the \(lcc\) translation \([P]\) is executed, i.e., \(\forall z \in \{z_1, z_2\} \rightarrow \{P_1, P_2, \ldots, P_n\}\). 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 \(\pi\) program \((\vdash P)\) (Not.3.3), then 
\[
[P] \equiv \exists \bar{x}. \bar{y}. ([R_1] \parallel \ldots \parallel [R_n] \parallel V)
\]
where \(n \geq 1\), \(V = \{x_1 y_1 \parallel \ldots \parallel x_n y_n \parallel \} \parallel \bar{x}, \bar{y}, \bar{c} \parallel \bar{c} \parallel \bar{c} \parallel [V]\). Note that each \(R_i\), \(1 \leq i \leq n\), is a pre-redex.

**Definition 4.10** (Continuation processes). Let \(P\) be an \(\pi\) process such that \(P \equiv (\nu \bar{x} \bar{y} \bar{z})([\pi \nu \bar{v} v.Q \parallel R] \rightarrow [Q].\) for some \(Q, R, \bar{x}, \bar{y}, \bar{c} \). Assume \(x_i \in \bar{x}, y_i \in \bar{y}, c_i \in \bar{c}\) are co-variables. The continuation process of \(P\), denoted \([P]_{\nu \bar{v} y}\), is defined as follows:

1. If \(P \equiv (\nu \bar{x} \bar{y} \bar{z})([\pi \nu \bar{v} v.Q \parallel R])\) then 
\[
[P]_{\nu \bar{v} y} \equiv \forall \bar{x}. \forall \bar{y}. \forall \bar{z}. [\pi \nu \bar{v} v.Q \parallel R].
\]
2. If \(P \equiv (\nu \bar{v} \bar{z})([\pi \nu \bar{z} Q \parallel R])\) then 
\[
[P]_{\nu \bar{v} y} \equiv \forall \bar{v}. \forall \bar{z}. [\pi \nu \bar{z} Q \parallel R].
\]

We write \([P]\) when the co-variable \(y\) is unimportant.

We may now define:

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

\[
[P] = \{C\}[[R_1], \ldots, [R_n], \ldots, [R_n]]
\]

with \(1 \leq i \leq n\). Let \(S\) be an \(lcc\) process such that 
\[
S = \{C\}[[R_1], \ldots, [R_n], \ldots, [R_n]], 1 \leq i \leq n
\]

We say \(S\) is an intermediate process of \([P]\), denoted \(S \in \{[P]\}\), if there exist \(S'\) and \(S''\) such that \([P] \equiv [S' \parallel S'' \equiv S]\). 

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 ensures dynamic properties for the transference of reasoning techniques from \(\pi\) to \(lcc\):

**Theorem 4.12** (Operational Correspondence for \([\pi]\)). Let \([\pi]\) be the translation in Def. 4.3. Also, let \(P, Q\) be well-typed \(\pi\) programs and \(R, S\) be \(lcc\) processes. Then:

1. **Soundness:** If \(P \rightarrow \pi\) \(Q\) then either:
   a. \([P] \rightarrow R\), such that \(R \equiv \{Q\}\).
   b. \((\text{or}) [P] \equiv S' \rightarrow \pi R' \equiv R, \text{for some } R', S', \text{such that } R \equiv \{Q\}\).

2. **Completeness:** If \(P \rightarrow \pi\) \(S\) then either:
   a. \(P \rightarrow \pi\) \(Q, \text{for some } Q \text{ and } \{Q\} \approx S\).
   b. \((\text{or}) S \in \{[P]\} \text{ and, for some } S' \text{ and } Q, \text{we have that } S \rightarrow \pi S', \rightarrow P \rightarrow \pi, \text{and } \{Q\} \approx S'.

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

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 \(\{[\pi]\}, \varphi_1\) is an encoding (cf. Def. 4.2).

### 5. A Type System for \(lcc\)

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

### 5.1 Linear Abstractions with Local Information

Abstractions in \([13]\) act on global information posted in the store. This may be an issue when dealing with processes that appeal to their local information to perform some observable (public) behavior. To remedy this, we consider a variant of \(lcc\) in which abstractions are generalized so as to account for local information:

\[
\forall \bar{x}. (d; e \rightarrow P)
\]

Intuitively, \(d\) is a piece of local information used jointly with \(e\) to trigger \(P\). Formally, we extend Fig. 8 with the following rule:

\[
(C: \text{SYNLOC})
\]

\[
e \odot d \vdash C \exists y (e \odot f) \quad y \odot f \vdash e (c, d, e, P) = 0
\]

\[
\text{msgc}(e \odot d, \exists y (e \odot f)) \quad c \odot d \vdash C 0 \Rightarrow e \vdash C 0
\]

\[
\exists \parallel \forall \bar{x}. (d; e \rightarrow P) \quad \exists \parallel \forall \bar{x}. (P \parallel f) \parallel \bar{x}
\]

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 \(e \odot d \vdash C 0 \Rightarrow e \vdash C 0\) 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 \(\pi\) into \(lcc\) introduced in §4.1 relies critically on abstractions to represent synchronizations in \(\pi\), 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
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 \( \pi \) process:

\[
S = (\nu x y)(\exists v. P_x \mid y(x). Q_y) \mid R
\]

Under the assumption that \( \text{fn}(P, Q) \cap \text{fn}(R) = \emptyset \), the restriction \( (\nu x y) \) 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 aholt of \( v \) in the reduction

\[
S \rightarrow_x (\nu x y)(P_x \mid Q_y[v/y]) \mid R
\]

Unfortunately, the privacy guarantees offered by restriction in \( \pi \) do not extend to \( \llcc \), which seriously hinders one of the main assumptions in session-based concurrency. Consider the \( \llcc \) process

\[
[(\nu x y)(\exists v. P_x \mid y(x). Q_y)] \gg A
\]

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

\[
A = \forall y, w(tt; out(w, y) \otimes \{x:w\} \rightarrow \text{in}(x, y) \gg Spy)
\]

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 \( [(\nu x y)(\exists v. P_x \mid y(x). Q_y)] \) 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 \( \pi \) processes are compiled down into \( \llcc \).

Note that this anomaly is not particular of our encoding \( [2] \); rather, it affects all \( \llcc \) processes that use abstractions to synchronize-like processes. Scope extensions as the one possible in \( [3] \) are clearly not possible in \( \pi \), and we must limit the power of abstractions so as to preserve the very nature of the restriction operator in \( \pi \). Intuitively, this means that the privacy of session endpoints must be explicitly programmed at the declarative level of \( \llcc \), relying on some extra mechanism that limits abstractions.

To this end, we rely on a simple typing discipline, built upon the approach in \( [13] \) (where the focus is in \( \llcc \) 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 we distinguish between two sorts of variables: 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 a simple access control mechanism for \( \llcc \) abstractions.

A well-typed \( \llcc \) process in our type system is a process in which all abstractions \( \forall \bar x(d;c \rightarrow P) \) are such that \( c \) 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 \( \pi^+ \) into \( \llcc \); see \( \S 6 \). 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 \( [(\nu x y)(\exists v. P_x \mid y(x). Q_y)] \) from process \( [(\nu x y)(\exists v. P_x \mid y(x). Q_y)] \gg A \); while the former is well-typed, the latter is not (see also Example \( \S 4 \)).

5.3 The Typing System

The typing rules for secure patterns/processes are defined in Fig. 8.

[Figure 8. Typing rules for \( \llcc \). Rule (L:ASSOC-R) is omitted.]

We employ 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 occurring in \( t \). Formally, these functions are given by:

\[
\begin{align*}
unr(x) &= res(x) = var(x) = \{x\} \ (x \text{ is a variable}) \\
unr(\gamma(t_1; t_2)) &= unr(t_2) \\
res(\gamma(t_1; t_2)) &= res(t_1) \\
var(\gamma(t_1; t_2)) &= var(t_1) \cup var(t_2)
\end{align*}
\]

We assume \( unr(x), res(x), \) and \( var(x) \) extend to vectors \( \bar x \) 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 \( \gamma(t_1; t_2) \), terms in \( t_2 \) could contain restricted variables (in nested predicates, for instance).

As hinted above, the objective of the type system is to identify \( \llcc \) processes whose abstractions contain secure patterns. We consider three kinds of judgments. Judgment \( \Delta; \Theta \vdash c \) concerns patterns: it says that pattern \( c \) is well-formed, under restricted variables \( \Delta \) and unrestricted variables \( \Theta \). The judgment for guards (abstractions, non-deterministic choice) is denoted \( \Delta; \Theta \vdash \gamma \) whereas a well-typed process \( P \) is denoted by \( \vdash P \).

We comment on typing rules in Fig. 8. Rules (L:ASSOC-L), (L:ASSOC-R), and (L:COMM) define basic properties of constraint conjunctions. Given a predicate \( \gamma(t_1; t_2) \), rule (L:PRED) decrees that all variables in \( t_1 \) as well as the variables occurring restricted in \( t_2 \) are restricted. The remaining variables are unrestricted. Rule (L:COMM) identifies the restricted and unrestricted variables in the pattern \( c \circ d \). 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 \( c \circ d \).

Typing rules for guards and processes are simple. The most interesting rule is (L:ABS), which says that abstraction \( \forall \bar x(d;c \rightarrow P) \) is secure as long as variables \( \bar x \) are unrestricted in the typing for \( c \), and no variables in \( d \) occur in \( \bar x \).

The main theorem regarding the type system is type preservation (Theorem \( \S 5.2 \)), whose proof relies on subject congruence:

Lemma 5.1. If \( P \equiv_i Q \) and \( \vdash P \), then \( \vdash Q \).
Theorem 5.2 (Type Preservation). If $P \xrightarrow{\alpha} Q$ and $\vdash_o P$ then $\vdash_o Q$.

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

$$A' = \forall y, w. (t t o u t(w, y) \otimes \{ w: x \}) \rightarrow n(x, y) \parallel [P])$$

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 malicious contexts. Using rule (L:C), 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 $\pi^+$ with Session Establishment

We now present our second encoding. The source calculus is $\pi^+$, the extension of $\pi$ with constructs for session establishment based on explicit locations. The target calculus is $\lambda cc$ with abstractions and local information. This second encoding builds upon the one given in § 3.1 to accommodate a secure phase of session establishment. We show that our extended encoding maintains the correctness properties of the encoding in § 3.1, and is well-typed in the discipline given in § 3.2. As such, our extended encoding enjoys a robust treatment of restriction and scope extrusion, ensured by combining generalized abstractions and secure patterns.

Next we introduce $\pi^+$ (§ 6.1), present its translation into $\lambda cc$ (§ 6.2), and establish that this translation is an encoding (cf. Def. 4.2) and is well-typed (§ 6.3).

6.1 The Calculus $\pi^+$

The syntax of $\pi^+$ extends Def. 3.1 with service requests and accepts, two constructs for representing session establishment. Our constructs extend those defined in § 4 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 $n, m, \ldots$ range over locations; also, let $\rho$ denote a set of locations. The syntax of $\pi^+$ extends $\pi$ with two constructs:

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

The operational semantics for $\pi^+$ extends the reduction relation given in Fig. 1 with the following rule (denoted [EST]):

$$[\pi^m(z), P]^n \rightarrow [a^\rho^m_\nu(x), Q]^m \rightarrow (uxyz)(P[y/z]) \parallel Q \quad (n \in \rho)$$

With a slight abuse of notation, we will write $\rightarrow_\pi$ to denote a reduction step in $\pi^+$. 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 $\pi$. This way, $\pi^+$ can be seen as being at a higher abstraction level than $\pi$. This convenience is useful for modeling, but it does not represent an expressiveness gain: we can represent service acceptance and request in $\pi$. Define the translation $[\pi^+] : \pi^+ \rightarrow \pi$ as

$$[[\pi^m(z), P]^n]^+ = a_1 \triangleleft m.a_1 \triangleleft a_1(z).[[P]^+]$$

$$[[a^\rho^m_\nu(x), Q]^m]^+ = a_2 \triangleright \{ m : a_2 \triangleright \{ l_i : (uxyz)(\overline{a_2 y} \parallel [[Q]^+]\}_i, \_\} $$

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

Proposition 6.1. Let $S = [[\pi^m(z), P]^n]^+$, $[[a^\rho^m_\nu(x), Q]^m]^+$ be an $\pi^+$ process, with $n \in \rho$, Then: If $S \rightarrow_\pi S'$ then $S' = (uxyz)(P[y/z] \parallel Q)$

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

6.2 Translating $\pi^+$ into $\lambda cc$

We now present a translation of $\pi^+$ into $\lambda cc$. Key novelties with respect to the encoding given in § 4 are: first, we consider the session establishment phase with locations; 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 abstractions with local information ($\pi^+$), 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 $\pi^+$ into $\lambda cc$ relies on a security constraint system that combines local and global information with basic cryptographic primitives. Building upon similar constraint systems in [15][24][24], we provide the following definition.

Definition 6.2 (Security Constraint System). Consider the tuple $(\Sigma, \Sigma, \triangleright\triangleright \subset \subset \cap \cap \vee \vee)$ where $\Sigma$ is the set of all constraints obtained by using linear operators $!$, $\otimes$ and $\exists$ over the function symbols of $\Sigma$ (Fig. 2), predicate $o(x)$, and where $\triangleright\triangleright \subset \subset \cap \cap \vee \vee \cap$ is given by the usual deduction rules for linear logic with syntactic equality and the rules in Fig. 2.

We briefly comment on the signature $\Sigma$ given in Fig. 2, which differs from that in Fig. 3 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 $o(x)$ stands for the output of message $x$ in some public medium (say, an unprotected network). Function enc returns the encrypted message using a key $x$. Functions p(x), s(x), and b(x) return the public, restricted (private), or symmetric keys of a channel $x$, respectively. Function tup allows us to create $n$-ary tuples (with $|L| \geq 1$). Given an unrestricted variable $x$ and a set $\rho$, function loc$_\rho(x)$ returns 1 if $x \in \rho$ and 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., $o(out(x;m))$), it is possible to read the message $m$. Rule (E:COV) relates the two endpoints, known

1. This choice is orthogonal to the translation; other, more sophisticated protocols could be considered.
\[ \Sigma \equiv \text{in}(x, y; e) \mid \text{out}(x; y) \mid \text{sel}(x; l) \mid \text{br}(x; l; e) \mid \text{enc}(x; y) \]
\[ \mid \text{covar}(x; y; e) \mid \text{tup}_n(\vec{x}) \mid \text{loc}_n(x) \mid p(x) \mid r(x) \mid s(x) \]

**Figure 9.** Security constraint system: Function symbols.

(E:OUTM) \[ c \vdash f \circ o(x) \quad c \vdash f \circ o(m) \]
\[ \text{if } \gamma \in \{\text{out}, \text{in}, \text{sel}, \text{br}\} \]

(E:COV) \[ c \vdash f \circ o(x) \quad c \vdash f \circ o(y) \]
\[ x \neq y \]

(E:KEY) \[ c \vdash f \circ o(x) \quad k \in \{s, p, r\} \]

(E:ENC) \[ c \vdash f \circ o(k(x)) \]

(E:DEC) \[ c \vdash f \circ o(m) \]

(E:TUP) \[ \forall j \in \{1, \ldots, n\}, c \vdash f \circ o(i_j) \]

(E:PROJ) \[ c \vdash f \circ o(t_{n}(i_1, \ldots, i_n)) \]

**Figure 10.** Security constraint system: Entailment relation.

only to the participants of that interaction: it states that given an endpoint key \(x\) and the co-variable constraint, we may 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) expresses 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 \(\text{enc}(x; y)\) will be written as \(\{y\}_x\). Also, tuple \(\text{tup}_n(x_1, \ldots, x_n)\), with \(n \geq 1\), will be written \(\{x_1, \ldots, x_n\}\).

### 6.2.2 The Translation

We now introduce a translation of \(\pi^+\) with secure session establishment into \(\perp\). As explained in §5, one of the challenges associated to a translation of session establishment is that the use of abstractions over constraints containing only unrestricted predicates enables any external process to abstract (private) session keys. To solve this issue, our translation of session establishment includes an explicit authentication protocol (the NSL protocol). The translation is defined as follows.

**Definition 6.4 (\(\pi^+\) into \(\perp\)).** We define the translation from \(\pi^+\) into \(\perp\) as the pair \(\langle [\cdot]^{\perp}, \varphi_1^{\perp} \rangle\), where:

(a) \([\cdot]^{\perp}\) is the process mapping defined in Fig. 11

(b) \(\varphi_1^{\perp}\) is defined as in Def. 5.3(b).

The translation in Fig. 11 is similar to the encoding of \(\pi^+\) into \(\perp\) (cf. Fig. 9). Two differences concern the authentication protocol and local information. First, session establishment is realized via 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 \((x, y)\) and receives a tuple with \(w\) and \(n\). Notice that such a tuple is not made publicly available, 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 piece of local information, therefore preventing interferences. Generated after session establishment, these constraints are collected in a set \(f\), a parameter of the translation. This is made explicit in the translation of \((x:y)\) in Fig. 11 with a minor abuse of notation, we write \(f_x\) to denote the co-variable of \(x\) recorded in \(f\). Also, 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 \([\cdot]^{\perp}\) : \(\pi^+\) into \(\perp\), in the sense of Def. 7.2. We mostly build upon the approach given in §5 and §6.4. The notion of evaluation context is as in Def. 7.7. We also establish typability of encoded processes (Thm. 6.9).

**Theorem 6.5 (Name invariance for \([\cdot]^{\perp}\)).** Let \(P\), \(\sigma\), and \(x\) be a typable \(\pi^+\) process, a substitution satisfying the renaming policy for \([\cdot]^{\perp}\) (Def. 6.4(b)), and a variable in \(\perp\), resp. Then \([P[\sigma]]^{\perp}\) = \([P]^{\perp}[\sigma]\), where \(\varphi_1^{\perp}(\sigma(x)) = \varphi_1^{\perp}(\sigma(x))\) and \(\sigma = \sigma'\).

We may also show that our translation is compositional with respect to restriction and parallel.
Theorem 6.6 (Compositionality of $[[\ ]_g]$). Let $P$ and $E[\cdot]$ be a typable $\pi$-process and an $\pi$-evaluation context (cf. Def. 4.7), respectively. Then we have: $[[E[P]]^{\cdot}_g] = [[E]^{\cdot}_g] [[P]^{\cdot}_g]$

We now state operational correspondence. Recall that notation $S \in \lcc$ has been introduced in Def. 4.11.

Theorem 6.7 (Operational Correspondence for $[[\ ]_f]$). Let $P, Q$ be typable $\pi$-programs and $R, S \in \lcc$ processes. Then:

1. Soundness: If $P \rightarrow_{\nu} Q$ then either:
   a. $[[P]^{\cdot}_f] \rightarrow_{\nu} R$, for some $R$ such that $R \equiv [[Q]^{\cdot}_f]^{\nu}$,
   b. (or) $[[P]^{\cdot}_f] \approx R$, for some $R$ such that $R \equiv [[Q]^{\cdot}_f]^{\nu}$.
   c. (or) $[[P]^{\cdot}_f] \rightarrow_{\nu} R$, for some $R$ such that $R \equiv [[Q]^{\cdot}_f]^{\nu}$.

2. Completeness: If $[[P]^{\cdot}_f] \rightarrow_{\nu} S$ then either:
   a. $P \rightarrow_{\nu} Q$, for some $Q$ and $[[Q]^{\cdot}_f] \equiv S$.
   b. (or) $S \in [[P]^{\cdot}_f]$ and, for some $S'$ and $Q$, we have that $S \rightarrow_{\nu} S', P \rightarrow_{\nu} Q$, and $[[Q]^{\cdot}_f] \equiv S'$.
   c. (or) $S \in [[P]^{\cdot}_f]$ and $S \rightarrow_{\nu} S'$, for some $S'$ and $Q$, we have that $P \rightarrow_{\nu} Q$ and $[[Q]^{\cdot}_f] \equiv S'$.

We discuss some differences with respect to the case of $\pi$. 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 $\pi^+$ 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 $P \equiv s t^{\nu}(z).P''$, then we have that $\forall x.\forall y.\forall \sigma :: (x) \rightarrow \rho :: (x)$. The definition of intermediate process (Def. 4.11) is the same.

Based on the above theorems, we may state:

Corollary 6.8. Translation $[[\ ]_f$, $\varphi[\ ]_f]$ is an encoding (Def. 4.2).

Our final correctness property for the translation is typability with respect to the type system in $S[\ ]$

Theorem 6.9 (Typability of $[[\ ]_f]$). Let $P$ be a typable $\pi$-process. Then $\vdash_\nu [[P]^{\cdot}_f]$

The proof of Theorem 6.9 is by induction on the structure of $P$. Fig. 7 gives the derivation tree for the case $P = [a(x).Q]^m$.

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 $[[\ ]_f$, 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 5.6 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

Our developments build upon our previous works [15, 19]. However, because of the substantial technical differences (notably, the presence of linearity) our results cannot be derived from those in [15, 19]. A key difference with respect to [19] is the $\pi$-calculus considered ($\lcc$ here, $\utcc$ in [19]); this is crucial because, as already discussed, thanks to the linear abstractions in $\lcc$, our encodings of $\pi$ and $\pi^+$, presented in [1] and [6], are rather compact and count with tight operational correspondences. We also improve on expressiveness: since $\utcc$ is a deterministic calculus, the encoding in [19] cannot capture non-deterministic behavior (as useful in session establishment). In contrast, exploiting linearity, our encoding captures non-deterministic session establishment and also forms of non-determinism derivable using unrestricted types in $\pi$. Fig 8 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 $\pi$. 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 [13] and the one in [5] are similar in spirit, but not in details: the language in [13] is $\utcc$, and moving to $\lcc$ and considering linearity requires non-trivial modifications.

Haemmerlé [14] gives an $\lcc$ encoding of an asynchronous $\pi$-calculus, and establishes its operational correspondence. Since his encoding concerns two asynchronous models, this operational correspondence is rather direct. Monjaraz and Marifio [22] encode the asynchronous $\pi$-calculus into Flat Guarded Horn Clauses. They consider compositionality and operational correspondence issues, as we do here. In contrast to [13, 22], here we consider a session $\pi$-calculus with synchronous communication, which adds challenges in the encoding and its associated correctness proofs.

The relationship between linear logic and session types has been recently established. CaIres and Pfennig gave an interpretation of intuitionistic linear logic as session types, in the style of Curry-Howard [4]. Wadler developed this interpretation for classical linear logic [28]. Grandi 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 specifications.

8. Concluding Remarks

We presented two encodings of session $\pi$-calculi into $\lcc$, a declarative process model based on the $\pi$-calculus paradigm. The encodings crucially exploit linearity, a common trait in both models. Linearity enables us to define intuitive translations of session-based processes and to obtain clean correctness properties for them (notably, operational correspondence), improving our previous work [19].

Our first encoding concerns $\pi$, the session $\pi$-calculus in [27]; the second encoding considers $\pi^+$, an extension of $\pi$ 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, 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 $\pi$-calculus.

In future work, we wish to explore whether reasoning techniques for $\lcc$ processes can support the analysis of $\pi$ processes, for instance, to ensure deadlock-freedom. Also, 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 thank Rémy Haemmerlé, Carlos Olarte, and Vasco Vasconcelos for useful exchanges, and to the anonymous reviewers for their valuable suggestions. This research was partially supported by the EU COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems, by FCT project Liveness, Statistically (PTDC/EIA-CCO/117513/2010), by the Danish Foundation for Basic Research, project IDEA4CPS (DNRF86-10) and by the Pontificia Universidad Javeriana, project D-Spaces. Pérez is also affiliated to NOVA Laboratory for Computer Science and Informatics, Universidade Nova de Lisboa, Portugal.

References
(T:BOOL) \Gamma \vdash ff, tt : q bool
\Gamma, x : T, \Gamma_2 \vdash x : T \quad \Gamma \vdash 0

(T:PAR) \Gamma_1 \vdash P \quad \Gamma_2 \vdash Q
\Gamma_1 \odot \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q
\Gamma \vdash (\nu xy)P

(T:IN) \Gamma_1 \vdash x : q?T.U \quad \Gamma_2, y : T \vdash o x : U \vdash P
\Gamma_1 \odot \Gamma_2 \vdash x : y.P

(T:OUT) \Gamma_1 \vdash x : qT.U \quad \Gamma_2 \vdash v : T \quad \Gamma_3 \vdash o x : U \vdash P
\Gamma_1 \odot \Gamma_2 \vdash x \odot \Gamma_3 \vdash T.v.P

(T:SEL) \Gamma_1 \vdash x : q \{ \{ l_i : T_i \} \}_{i \in I} \quad \forall i \in I, \Gamma_2 \vdash x : T_i \vdash P_i
\Gamma_1 \odot \Gamma_2 \vdash x \odot \Gamma_3 \vdash \Gamma_i \odot \Gamma_3 \vdash T_i.v.P

(T:BRA) \Gamma_1 \vdash x : q \{ \{ l_i : T_i \} \}_{i \in I} \quad \forall i \in I, \Gamma_2 \vdash x : T_i \vdash P_i
\Gamma_1 \odot \Gamma_2 \vdash x \odot \Gamma_3 \vdash \Gamma_i \odot \Gamma_3 \vdash T_i.v.P

(T:REPL) \Gamma \vdash P : \nu \{ \{ x \} \}_{i \in I} \quad \Gamma_1 \vdash \nu \{ \{ x \} \}_{i \in I} : \nu \{ \{ x \} \}_{i \in I}
\Gamma \vdash \nu \{ \{ x \} \}_{i \in I} : \nu \{ \{ x \} \}_{i \in I}

A. Omitted Definitions

A.1 Context Splitting

Definition A.1 (Context splitting). Let \Gamma_1, \Gamma_2 denote concatenation of context \Gamma_1 \odot \Gamma_2. Context splitting is defined as follows:
\Gamma = \Gamma_1 \odot \Gamma_2
\Gamma_1 \vdash x : T = (\Gamma_1, x : T) \odot (\Gamma_2, x : T)
\Gamma_1 \odot \Gamma_2 \vdash x : T = (\Gamma_1 \odot \Gamma_2, x : T)

A.2 Complete Set of Typing Rules for \pi\sigma

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(\Gamma). 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 the proofs for the static properties of the encoding.

Theorem 4.8 (Compositionality of \pi\sigma). Statement on page 5

Proof. The proof proceeds by induction on the structure of \pi\sigma and a case analysis on the grammar in Def. 4.7.

• Subcase E[\cdot] = R. Analogous to previous one.

• Subcase E[\cdot] = \nu \{ x \}(). By Fig. 5 and structural congruence see that (\nu x.y)(0) = 0, thus \E[0] = E[0].

Case P = \pi v.Q:

• Subcase E[\cdot] = R \cdot

• Subcase E[\cdot] = \nu \{ x \}().

All the other cases proceed exactly in the same way as the previous one.

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 \pi\sigma. Case Rule [\nu \{ x \}]:

(i) Assume P = tt? (P')?(P'')

(ii) By (i) then \pi \rightarrow \pi' using [\nu \{ x \}].

(iii) Then by the application of the definition of [\cdot] (Def. 4.5):

\P = \nu \{ t t = t t \rightarrow [P'] \}

\nu \{ t t = f f \rightarrow [P''] \}

(iv) By using rule (C:SYNC) (Fig. 5), with c = 1 we have that (note that \nu t t = t t)

\P \rightarrow [P'] \parallel \nu \{ t t = f f \rightarrow [P''] \}

(v) By (iv) note that the process

\nu \{ t t = f f \rightarrow [P''] \}

is blocked, the constraint tt = ff cannot be satisfied. Then, conclude that [P'] = \approx [P] \parallel \nu \{ t t = f f \rightarrow [P''] \}.

Case [\nu \{ x \}]: Analogous to previous case.

Case [\nu \{ x \}]:

(i) Assume P = (\nu x.y)(\pi v.P' \mid y(z).P'')
(ii) By (i) \( P \rightarrow_{\text{ref}} \nu xy \nu(x',y) \) using \([\text{COM}].\)

(iii) By definition of \([\cdot]::\):
\[
\begin{align*}
\llbracket P \rrbracket & = \exists x, y, ((\llbracket x : y \rrbracket \parallel \llbracket \text{out}(x, v) \rrbracket \parallel \\
& \quad \forall z((\text{in}(z, v) \otimes \{x : z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\
& \quad \forall z, w(\text{out}(w, z) \otimes \{w : y\}) \rightarrow \\
& \quad (\llbracket \text{in}(y, z) \rrbracket \parallel \llbracket P'' \rrbracket))
\end{align*}
\]

(iv) By using the rules of structural congruence and reduction of \( 1_{cc} \) one can build the following reduction:
\[
\begin{align*}
\llbracket P \rrbracket & \equiv_{\text{t}} \exists x, y, ((\llbracket x : y \rrbracket \parallel \llbracket \text{out}(x, v) \rrbracket \parallel \\
& \quad \forall z((\text{in}(z, v) \otimes \{x : z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\
& \quad \forall z, w(\text{out}(w, z) \otimes \{w : y\}) \rightarrow \\
& \quad (\llbracket \text{in}(y, z) \rrbracket \parallel \llbracket P'' \rrbracket))
\end{align*}
\]

(v) Conclude by considering the form of the process obtained in the previous derivation as follows:
\[
\begin{align*}
\llbracket (\nu xy)(P' \parallel P'')(\{v/z\}) \rrbracket & = \\
\exists x, y, ((\llbracket x : y \rrbracket \parallel \llbracket P' \rrbracket \parallel \llbracket P'' \rrbracket(\{v/z\}))
\end{align*}
\]

Case Rule [REPL.]: Analogous to case \([\text{COM}].\) as follows:

(i) Assume \( P = (\nu xy)(\exists v.P' \parallel \nu y(z).P'').\)

(ii) By (i) \( P \rightarrow_{\text{ref}} (\nu xy)(P' \parallel P'')(\{v/z\}) \parallel \nu y(z).P'\) using \([\text{REP}].\)

(iii) By definition of \([\cdot]::\):
\[
\begin{align*}
\llbracket P \rrbracket & = \exists x, y, ((\llbracket x : y \rrbracket \parallel \llbracket \text{out}(x, v) \rrbracket \parallel \\
& \quad \forall z((\text{in}(z, v) \otimes \{x : z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\
& \quad \forall z, w(\text{out}(w, z) \otimes \{w : y\}) \rightarrow \\
& \quad (\llbracket \text{in}(y, z) \rrbracket \parallel \llbracket P'' \rrbracket))
\end{align*}
\]

(iv) Let
\[
R = \forall z, w(\text{out}(w, z) \otimes \{w : y\}) \rightarrow (\llbracket \text{in}(y, z) \rrbracket \parallel \llbracket P'' \rrbracket)
\]

Then, by using the rules of structural congruence and reduction of \( 1_{cc} \) one can build the following reduction:
\[
\begin{align*}
\llbracket P \rrbracket & \equiv_{\text{t}} \exists x, y, ((\llbracket x : y \rrbracket \parallel \llbracket \text{out}(x, v) \rrbracket \parallel \\
& \quad \forall z((\text{in}(z, v) \otimes \{x : z\}) \rightarrow \llbracket P' \rrbracket) \parallel \\
& \quad \forall z, w(\text{out}(w, z) \otimes \{w : y\}) \rightarrow \\
& \quad (\llbracket \text{in}(y, z) \rrbracket \parallel \llbracket P'' \rrbracket)) \parallel R
\end{align*}
\]
process, which is treated as in case [Com] of the completeness result.

Subcase \( R_1 = x \triangleq l \cdot R'_1 \) and \( R_2 = x \triangleright \{ l : P_i \}_{i \in I} \) 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).

\( \square \)

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 \( \bar{t} = t_1, \ldots, t_n \) and process variables \( \bar{x} = x_1, \ldots, x_m \), the application of a substitution to a constraint, guard and process, denoted respectively \( c \{ \bar{y} \}, \{ \bar{y} \} \), is inductively defined on the structure of constraints, guards and process as:

\[
\begin{align*}
1 \{ \bar{y} \} & \overset{\text{def}}{=} 1 \\
0 \{ \bar{y} \} & \overset{\text{def}}{=} 0 \\
\gamma(\bar{v} ; \bar{v}) \{ \bar{y} \} & \overset{\text{def}}{=} \gamma(\bar{v} ; \bar{v} \{ \bar{y} \} ) \\
(c \otimes d) \{ \bar{y} \} & \overset{\text{def}}{=} c \{ \bar{y} \} \otimes d \{ \bar{y} \} \\
(\exists \bar{a} \cdot c) \{ \bar{y} \} & \overset{\text{def}}{=} \exists \bar{a} \cdot c \{ \bar{y} \} (\bar{y} \cap \bar{x} = 0) \\
\langle a \rangle \{ \bar{y} \} & \overset{\text{def}}{=} \langle a \rangle \{ \bar{y} \} \\
(\forall \bar{w} \cdot P) \{ \bar{y} \} & \overset{\text{def}}{=} \forall \bar{w} (\bar{w} \{ \bar{y} \} \cdot c (\{ \bar{y} \}) \rightarrow P(\{ \bar{y} \})) (\bar{y} \cap \bar{x} = 0) \\
(G_1 + G_2) \{ \bar{y} \} & \overset{\text{def}}{=} G_1 \{ \bar{y} \} + G_2 \{ \bar{y} \} \\
(\exists \bar{y} \cdot \forall \bar{x} \cdot Q) \{ \bar{y} \} & \overset{\text{def}}{=} \exists \bar{y} \cdot \forall \bar{x} \cdot Q(\bar{y} \{ \bar{y} \} ) \\
(\exists \bar{y} \cdot P) \{ \bar{y} \} & \overset{\text{def}}{=} \exists \bar{y} \cdot P \{ \bar{y} \} (\gamma \cap \bar{x} = 0) \\
(\exists \bar{y} \cdot P) \{ \bar{y} \} & \overset{\text{def}}{=} \exists \bar{y} \cdot P \{ \bar{y} \} (\gamma \cap \bar{x} = 0) \\
(G) \{ \bar{y} \} & \overset{\text{def}}{=} G \{ \bar{y} \} \\
(\exists \bar{y}, \exists \bar{x} \cdot P) & \overset{\text{def}}{=} \exists \bar{y}, \exists \bar{x} \cdot P \\
\end{align*}
\]

Lemma 5.1 (Subject Congruence). Statement on page 8

Proof. The proof proceeds by induction on the depth of the premise \( P \equiv_1 Q \).

Case

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule premise</td>
<td>( P \parallel \bar{I} \equiv_1 P )</td>
<td>(1)</td>
</tr>
<tr>
<td>Rule premise</td>
<td>( \vdash \circ \bar{I} \parallel P )</td>
<td>(2)</td>
</tr>
<tr>
<td>2, inversion</td>
<td>( \vdash \circ \bar{I} \parallel P )</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Case

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule premise</td>
<td>( \exists \bar{z} \cdot \bar{I} \equiv_1 \bar{I} )</td>
<td>(1)</td>
</tr>
<tr>
<td>Rule premise</td>
<td>( \vdash \circ \exists \bar{z} \cdot \bar{I} )</td>
<td>(2)</td>
</tr>
<tr>
<td>2, inversion</td>
<td>( \vdash \circ \exists \bar{z} \cdot \bar{I} )</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Case

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule premise</td>
<td>( \exists \bar{x} \cdot \exists \bar{y} \cdot P \equiv_1 \exists \bar{y} \cdot \exists \bar{x} \cdot P )</td>
<td>(1)</td>
</tr>
<tr>
<td>Rule premise</td>
<td>( \vdash \circ \exists \bar{x} \cdot \exists \bar{y} \cdot P )</td>
<td>(2)</td>
</tr>
<tr>
<td>2, inversion</td>
<td>( \vdash \circ \exists \bar{x} \cdot \exists \bar{y} \cdot P )</td>
<td>(3)</td>
</tr>
<tr>
<td>3, inversion</td>
<td>( \vdash \circ \exists \bar{x} \cdot \exists \bar{y} \cdot P )</td>
<td>(4)</td>
</tr>
<tr>
<td>4, formation</td>
<td>( \vdash \circ \exists \bar{x} \cdot \exists \bar{y} \cdot P )</td>
<td>(5)</td>
</tr>
</tbody>
</table>

Proposition C.2. Let \( P \) and \( t \) be a process and a term, respectively. If \( \vdash \circ P \) then \( \vdash \circ P \{ t / y \} \).

Proof. By induction on the structure of \( P \). Interesting cases are when \( P = \bar{a} \) and \( P = \forall \bar{y} (d; e \rightarrow P') \), for some \( \bar{y}, d, e, \) and \( P' \); other cases are easy.
• Case \( P = \varepsilon \): By the well-typedness assumption we infer that \( c \in C \), which immediately implies that \( \{ \xi x \} \in C \) and so we conclude using (L:TELL).

• Case \( P = \forall y (d; e \rightarrow P') \): By the well-typedness assumption we infer that \( P' \) is well-typed, that \( \Delta; \Theta \vdash \cdot e \), and that \( \vec{y} \subseteq dom(\Theta) \setminus \text{fv}(d) \). Since universal and existential quantifiers are binders, occurrences of variables in \( \vec{y} \) are not affected by \( \{ \xi x \} \); 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.

\[ \square \]

**Theorem 5.2 (Type Preservation).** Statement on page 8

**Proof.** The proof proceeds by induction on the depth of the premise \( P \rightarrow Q \).

**Case (C:OUT)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:OUT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} )</td>
<td>( \bar{e} )</td>
<td>(1)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>2, 6, transitivity</td>
<td>2, 6, transitivity</td>
<td>(7)</td>
</tr>
<tr>
<td>7, formation</td>
<td>7, formation</td>
<td>(8)</td>
</tr>
</tbody>
</table>

**Case (C:SynLoc)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:SynLoc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(4)</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>Hypothesis</td>
<td>(5)</td>
</tr>
<tr>
<td>1, formation</td>
<td>1, formation</td>
<td>(6)</td>
</tr>
<tr>
<td>6, inversion</td>
<td>6, inversion</td>
<td>(7)</td>
</tr>
<tr>
<td>8, inversion</td>
<td>8, inversion</td>
<td>(8)</td>
</tr>
<tr>
<td>9, inversion</td>
<td>9, inversion</td>
<td>(9)</td>
</tr>
<tr>
<td>9, inversion</td>
<td>9, inversion</td>
<td>(10)</td>
</tr>
<tr>
<td>9, inversion</td>
<td>9, inversion</td>
<td>(11)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(12)</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>Hypothesis</td>
<td>(13)</td>
</tr>
<tr>
<td>13, transitivity</td>
<td>13, transitivity</td>
<td>(14)</td>
</tr>
<tr>
<td>14, formation</td>
<td>14, formation</td>
<td>(15)</td>
</tr>
<tr>
<td>4, def. mgc</td>
<td>4, def. mgc</td>
<td>(16)</td>
</tr>
<tr>
<td>16, 17, Prop. C.2</td>
<td>16, 17, Prop. C.2</td>
<td>(17)</td>
</tr>
<tr>
<td>18, formation</td>
<td>18, formation</td>
<td>(18)</td>
</tr>
</tbody>
</table>

**Case (C:IN)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:IN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, well-formedness of labels</td>
<td>1, well-formedness of labels</td>
<td>(2)</td>
</tr>
<tr>
<td>2, inversion</td>
<td>2, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>1, well-formedness of labels</td>
<td>1, well-formedness of labels</td>
<td>(4)</td>
</tr>
<tr>
<td>4, formation</td>
<td>4, formation</td>
<td>(5)</td>
</tr>
</tbody>
</table>

**Case (C:COMP)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:COMP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>Hypothesis</td>
<td>(4)</td>
</tr>
<tr>
<td>6, 7, formation</td>
<td>6, 7, formation</td>
<td>(5)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:SUM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>3, inversion</td>
<td>3, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>5, inversion</td>
<td>5, inversion</td>
<td>(4)</td>
</tr>
<tr>
<td>6, inversion</td>
<td>6, inversion</td>
<td>(5)</td>
</tr>
<tr>
<td>7, formation</td>
<td>7, formation</td>
<td>(6)</td>
</tr>
<tr>
<td>8, formation</td>
<td>8, formation</td>
<td>(7)</td>
</tr>
</tbody>
</table>

**Case (C:EXT)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:EXT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>2, inversion</td>
<td>2, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>3, 4, IH</td>
<td>3, 4, IH</td>
<td>(4)</td>
</tr>
</tbody>
</table>

**Case (C:CONG)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:CONG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(4)</td>
</tr>
<tr>
<td>2, 3, lemma C.2</td>
<td>2, 3, lemma C.2</td>
<td>(5)</td>
</tr>
<tr>
<td>4, 6, IH</td>
<td>4, 6, IH</td>
<td>(6)</td>
</tr>
<tr>
<td>5, 7, lemma C.3</td>
<td>5, 7, lemma C.3</td>
<td>(7)</td>
</tr>
</tbody>
</table>

**Case (C:RES)**

<table>
<thead>
<tr>
<th>Rule/Premise</th>
<th>Rule/Premise</th>
<th>Case (C:RES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{e} | \forall \bar{e} \bar{c} \bar{d} \bar{e} \rightarrow P \rightarrow \exists \bar{y}. (P { \xi x } \bar{c} \bar{d} \bar{e} \rightarrow P) )</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(2)</td>
</tr>
<tr>
<td>1, inversion</td>
<td>1, inversion</td>
<td>(3)</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>Hypothesis</td>
<td>(4)</td>
</tr>
<tr>
<td>4, formation</td>
<td>4, formation</td>
<td>(5)</td>
</tr>
<tr>
<td>6, formation</td>
<td>6, formation</td>
<td>(6)</td>
</tr>
</tbody>
</table>

\[ \square \]
D. Appendix to Section [6]

Theorem 6.5 (Name Invariance for \( A^p \)). Statement on page 10

Proof. The proof for this theorem proceeds by induction on the structure of \( A \) as follows:
Case \( A = 0 \):
(i) By Fig. 11 we have that \([0] = T\)
(ii) Since there are no variables in \( T \) then \( T \sigma = T\)
(iii) By (i)(ii) we have that \([0 \sigma] = [0] \sigma\)
Case \( A = \pi v.Q \):
(i) By Fig. 11 and Def. 6.4(b) we have that:
\[ [\pi v.Q \sigma] = o(out(x, v) \sigma) \]
\[ \forall e((o(x) \otimes (x:f_x)) \sigma ; in(f_x, v) \sigma \rightarrow [Q \sigma]) \]
(ii) By Fig. 11 and Def. 6.4(b) we have that:
\[ [\pi v.Q \sigma] = o(out(x, v) \sigma) \]
\[ \forall e((o(x) \otimes (x:f_x)) \sigma ; in(f_x, v) \sigma \rightarrow [Q \sigma]) \]
(iii) By the Inductive Hypothesis we have that \([Q \sigma] = [Q] \sigma\)
(iv) By (i),(ii),(iii) we have that \([\pi v.Q \sigma] = [\pi v.Q] \sigma\)

All the other cases proceed exactly in the same way as the previous one.

Theorem 6.6 (Compositionality of \( A^p \)). Statement on page 10

Proof. The proof proceed by induction on the structure of \( A \) and a case analysis on the grammar in Def. 4.7.
Case \( A = 0 \):
* Subcase \( E = R \) :
  (i) By Fig. 11 \([E[0]] = [R] \| [0] \]
  (ii) By (i) and Fig. 11 we have that \([R] \| [0] = [E] \| [0] = [E[0]] \]
* Subcase \( E = \pi v.Q \) :
  (i) By Fig. 11 \([E[Q]] = [R] \| [Q] \]
  (ii) By (i) and Fig. 11 we have that \([R] \| [Q] = [E] \| [Q] = [E[Q]] \]
  (iii) By the Inductive Hypothesis we have that \([Q] = [Q] \]
  (iv) By (i),(ii),(iii) we have that \([E[Q]] = [E] \]

Case \( A = \pi v.Q \):
* Subcase \( E = R \) :
  (i) By Fig. 11 \([E[R]] = [R] \| [R] \]
  (ii) By (i) and Fig. 11 we have that \([R] \| [R] = [E] \| [R] = [E[R]] \]
  (iii) By the Inductive Hypothesis we have that \([R] = [R] \]
  (iv) By (i),(ii),(iii) we have that \([E[R]] = [E] \]

All the other cases proceed in the same way as the previous case.

Theorem 6.7 (Operational Correspondence for \( A^p \)). 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:
     \[ ([\pi^I (x).P^I] \| [a_o^H (z).Q]^I) \rightarrow_{e} (\nu^{z/y})(P^I \{y/x\} \mid Q) \]
     Where \( l_i \in \rho\)
     (ii) By the definition of the encoding:
     \[ ([\pi^I (x).P^I] \| [a_o^H (z).Q]^I) \rightarrow_{e} (\nu^{z/y})(P^I \{y/x\} \mid Q) \]
     \[ \rightarrow_{e} \exists \alpha, \beta, \gamma, \delta \in p, (P^I \{y/x\} \mid Q) \]
     (iii) Since \( m \) is a new variable that does not exists in \( P \) or \( Q \), conclude:
     \[ \exists \alpha, \beta, \gamma, \delta \in p, (P^I \{y/x\} \mid Q) \]
   
   • For the sake of illustration, we show the new structure for rule \([\text{COM}] \), note that \( \{x,y\} \in f \); thus \( f_x = y, f_y = x \). All the other cases proceed similarly.
     (i) Assume \( P = (\nu^{z/y})(P^I \mid y(z).P^I) \)
     (ii) By \( I \rightarrow_{e} (\nu^{z/y})(P^I \mid P^I \{y/z\}) \) using \([\text{COM}]\).
     (iii) By definition of \([I] : \)
     \[ [P^I] = \exists \alpha, \beta, \gamma, \delta \in p, (P^I \mid Q) \]
     \[ \rightarrow_{e} \exists \alpha, \beta, \gamma, \delta \in p, (P^I \mid Q) \]
     \[ \rightarrow_{e} \exists \alpha, \beta, \gamma, \delta \in p, (P^I \mid Q) \]
   
   • 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:
     \[ ([\pi^I (x).P^I] \| [a_o^H (z).Q]^I) \rightarrow_{e} (\nu^{z/y})(P^I \{y/x\} \mid Q) \]
     Where \( l_i \in \rho\)
     (ii) By the definition of the encoding:
     \[ ([\pi^I (x).P^I] \| [a_o^H (z).Q]^I) \rightarrow_{e} (\nu^{z/y})(P^I \{y/x\} \mid Q) \]
     (iii) Since \( m \) is a new variable that does not exists in \( P \) or \( Q \), conclude:
     \[ \exists \alpha, \beta, \gamma, \delta \in p, (P^I \{y/x\} \mid Q) \]
     \[ \rightarrow_{e} \exists \alpha, \beta, \gamma, \delta \in p, (P^I \{y/x\} \mid Q) \]
     (iv) By using the rules of structural congruence and reduction of \( \ell e \) one can build the following reduction:
     \[ [P^I] = \exists \alpha, \beta, \gamma, \delta \in p, (P^I \mid Q) \]
     \[ \rightarrow_{e} \exists \alpha, \beta, \gamma, \delta \in p, (P^I \mid Q) \]
     \[ \rightarrow_{e} \exists \alpha, \beta, \gamma, \delta \in p, (P^I \mid Q) \]
(v) Conclude by considering the form of the process obtained in the previous derivation as follows:

\[ (\nu x y)(P' \parallel P''(\nu/z))_{K} = \exists x, y. ((\nu x y) \parallel [P']_{K} \parallel [P''(\nu/z)]_{K}) \]

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:

\[ P = ([((\nu z\langle x \rangle).P)_{l_{1}}]_{m} \parallel [a_{v}(z).Q]_{l_{2}})]_{n} K_{S}f = \exists m. (o(R) \parallel \forall x (o(R) \rightarrow o(R)) \parallel \exists z, y. (o(R)) \parallel \forall w, l (o(R) \parallel \forall \epsilon (o(R) \rightarrow o(R))) \]

It is possible to see (by building a similar reduction for the case of soundness) that in one reduction we will have process \( S \in \langle [P]_{K} \rangle \), and that in 2 reductions and one structural congruence step we reach \( [Q]_{K} \) as desired.

\[ \square \]

**Theorem 6.9 (Well-typedness for \([J]_{Kf}\)).** Statement on page 11.

**Proof.** The proof proceeds by induction on the structure of \( P \).

- Case \( P = [\pi m\langle x \rangle.Q]_{n}^n \): The derivation tree is given in Fig. 14.
- Case \( P = [a_{v}(x).Q]_{m}^m \): The derivation tree is given in Fig. 12.
- Case \( P = x v.Q \): The derivation tree is given in Fig. 14.
- Case \( P = x r\{I : Q\}_{i} \in l \): The derivation tree is given in Fig. 16.
- Case \( P = x \triangleright \{I : Q\}_{i} \in l \): The derivation tree is given in Fig. 17.

Note that when using the inductive hypothesis, we previously need to use \( n \) steps with \((L:PAR)\).

**Case \( P = v?\langle Q \rangle : (R) \):** The proof is trivial as all the variables of an equality are unrestricted; we omit the derivation tree.

\[ \square \]
\[
\begin{align*}
\vdash \varphi \frac{\text{(IH)}}{\text{L:TELL}} \quad \vdash o((x)_{p(m)}) \quad \vdash o(x) \frac{\text{(L:PRED)}}{\text{L:TELL}} \quad \vdash o((x)_{p(m)}) \quad \vdash o(x) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((w, n)_{p(m)}) \quad \vdash o(x) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((w, n)_{p(m)}) \quad \vdash o(x) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash O \frac{\text{L:LOCAL}}{\text{L:GUARD}} \\
\end{align*}
\]

**Figure 14.** Typing derivation for \([\pi^m(x).Q]^n_f\).

\[
\begin{align*}
\vdash \{x, v\} \frac{\text{(IH)}}{\text{L:TELL}} \quad \vdash o((x)_v) \quad \vdash o(x) \frac{\text{(L:PRED)}}{\text{L:TELL}} \\
\vdash o((x)_v) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x)_v) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x)_v) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\end{align*}
\]

**Figure 15.** Typing derivation for \([\tau v.Q]_f^\circ\).

\[
\begin{align*}
\vdash o((x).Q) \frac{\text{(IH)}}{\text{L:TELL}} \quad \vdash o((x).Q) \frac{\text{(L:PAR)}}{\text{L:GUARD}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\end{align*}
\]

**Figure 16.** Typing derivation for \([x(y).Q]_f^\circ\).

\[
\begin{align*}
\vdash o((x).Q) \frac{\text{(IH)}}{\text{L:TELL}} \quad \vdash o((x).Q) \frac{\text{(L:PAR)}}{\text{L:GUARD}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\end{align*}
\]

**Figure 17.** Typing derivation for \([x l.Q]_f^\circ\).

\[
\begin{align*}
\vdash o((x).Q) \frac{\text{(IH)}}{\text{L:TELL}} \quad \vdash o((x).Q) \frac{\text{(L:PAR)}}{\text{L:GUARD}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\vdash o((x).Q) \frac{\text{L:GUARD}}{\text{L:PAR}} \\
\end{align*}
\]

**Figure 18.** Typing derivation for \([x \{l_i : Q_i\}_{i \in I}]_f^\circ\).