Abstract

Session-based concurrency is a type-based approach to the analysis of communication-intensive systems. These complex systems may be specified in an operational or declarative style: the former defines how interactions are properly structured; the latter defines governing conditions for correct interactions. In this paper, we investigate the relationship between operational and declarative models of session-based concurrency. We propose several interpretations (language encodings) of session $\pi$-calculus processes as declarative processes in linear concurrent constraint programming (1cc), a model of concurrency based on partial information (constraints). Our interpretations enjoy precise correctness properties, and offer a sound basis on which both operational and declarative requirements can be specified and reasoned about. We demonstrate that our encodings enable the bidirectional transfer of reasoning techniques between declarative and operational process models. Moreover, by coupling our interpretations with a type system for 1cc process, we obtain robust declarative encodings of $\pi$-calculus mobility.

Keywords: concurrency theory, process calculi, $\pi$-calculus, concurrent constraint programming, session types, expressiveness.
# Contents

1. Introduction .......................................................... 3

2. Overview of Key Ideas ............................................... 5

3. Preliminaries .......................................................... 10
   3.1 The session $\pi$-calculus ($s\pi$) .......................... 10
   3.2 Linear Concurrent Constraint Programming (lcc) ....... 15

4. Encoding $s\pi$ in lcc .................................................. 18
   4.1 The Notion of Encoding ....................................... 18
   4.2 Translating $s\pi$ into lcc ................................... 19
   4.3 Correctness of the Translation .............................. 21

5. Transfer of Techniques Between $s\pi$ and lcc ............... 24
   5.1 An Observational Equivalence for $s\pi$ ................. 25
   5.2 A Deadlock-Free Fragment of lcc ......................... 28

6. An Extension of lcc and Its Type System .................... 34
   6.1 lcc$^+$: lcc with Linear Abstractions with Local Information 34
   6.2 A Type System for lcc$^+$: Motivation .................... 34
   6.3 The Type System .............................................. 35

7. Encoding $s\pi$ with Session Establishment ................. 37
   7.1 The Calculus $s\pi^+$ ........................................ 38
   7.2 Translating $s\pi^+$ into lcc$^+$ ......................... 40
   7.3 Correctness of the Translation ............................ 42

8. Related Work ........................................................ 45

9. Concluding Remarks ................................................ 47

A. Appendix to Section 4 .............................................. 53
   A.1 Static Correctness of $\llbracket J \rrbracket_{s\pi}$ .......... 53
   A.2 Dynamic Correctness of $\llbracket J \rrbracket_{s\pi}$ ....... 54

B. Appendix to Section 5 .............................................. 57
   B.1 Appendix to Section 5.1 .................................... 57
   B.2 Appendix to Section 5.2 .................................... 65

C. Appendix to Section 6 .............................................. 68

D. Appendix to Section 7 .............................................. 75
   D.1 Preliminaries .................................................. 75
   D.2 Correctness for $\llbracket J \rrbracket_{lcc}$ ................. 76
   D.3 Correctness for $\llbracket J \rrbracket_{lcc^+}$ .............. 78
   D.4 Well-typedness for $\llbracket J \rrbracket_{lcc^+}$ .......... 81
1 Introduction

This paper relates two distinct models of concurrent processes: one of them, the session $\pi$-calculus \cite{Vas12} ($s\pi$ in the following), is operational; the other one, linear concurrent constraint programming \cite{FRS01,Hae11} (1cc in the following), 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 results of relative expressiveness, which describe how two different programming models can correctly represent each other. In turn, these expressiveness results offer a sound basis for the transfer of reasoning and validation techniques between the models. In this work, the common trait supporting the expressiveness results between $s\pi$ and 1cc is linearity, in the sense of linear logic—the well-known logic of consumable resources \cite{Cur87}.

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 \cite{HVK98}, against which specifications may be checked. When these specifications are given in the $\pi$-calculus \cite{MPW92}, we obtain 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 and session protocols are established. The second phase occurs when two dual partners interact according to the agreed session protocol. In this phase, interactions always occur in matching pairs: when one partner sends, the other receives; when one partner offers a selection, the other chooses; when a partner closes the session, the other must acknowledge. Sessions thus combine forms of concurrency and mobility, but also concern 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. Indeed, the connection between session-based concurrency and linear logic has been clarified via Curry-Howard correspondences \cite{CP10,Wad12}.

In describing structured communications, operational and declarative paradigms 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 languages based on the $\pi$-calculus are convenient to specify mobile, point-to-point communications, they do not satisfactorily express other kinds of requirements that influence interaction. In particular, partial and contextual information influencing protocols and/or partners can be unnatural or too convoluted to express in the $\pi$-calculus. Addressing this shortcoming of operational models, previous works have extended name-passing calculi with declarative features and constructs; see, e.g., \cite{DRV98,BM07a,CD09,BJPV11,BM11}. On the other hand, declarative models of concurrency naturally express partial and contextual information; see, e.g., \cite{dBGM00,FRS01,NPV02,OV08a}. Although some of these models (e.g., \cite{LM92,OV08a}) can represent mobility as in the $\pi$-calculus, such representations are indirect and so they may be unpractical for modeling and reasoning purposes.

In our view, all of the above begs for a unifying account of operational and declarative approaches to session-based concurrency. We envision a declarative basis for session-based concurrency which combines the best features from both paradigms: in such a unified basis, structuring constructs from operational models such as the $\pi$-calculus are given (correct) low-level declarative representations, which can be used as “macros” that can be freely composed with declarative constructs that express partial and contextual information. A declarative basis will in this way enable to articulate existing languages and analysis techniques at appropriate abstraction levels. In this work, we provide such a unifying account by formally relating different process languages; the resulting formal relations should in turn enable the sound transfer of verification techniques across operational and declarative models.
In a previous work [LOP10] some of the current authors described a preliminary step towards such a unified account of operational and declarative approaches. That work developed an encoding (language translation) of the session π-calculus in [HVK98] into declarative processes in universal concurrent constraint programming (utcc) [OV08a], a deterministic process language based on the concurrent constraint programming (ccp) model [Sar93]. Although the encoding in [LOP10] already enables us to reason about communication-intensive systems from a unified declarative standpoint, it has two limitations:

(a) The key role of linearity in session-based concurrency is not explicit in the encoding of session typed processes in utcc.

(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.

With the aim of addressing these shortcomings, in this paper we investigate new encodings of $\pi$ processes as processes in lcc, exploring also how they enable the transfer of reasoning techniques across operational and declarative settings. More precisely, in this paper we develop four technical contributions:

- To address (a), our first contribution is an encoding from $\pi$ processes into lcc. This encoding follows the approach in [LOP10] but, as an immediate consequence of targeting lcc instead of utcc, provides a direct treatment of linearity, as essential in operational approaches to session-based concurrency. Furthermore, we give precise operational correspondence results of the connection of both paradigms, following Gorla’s criteria for valid encodings [Gor10].

- Our second contribution takes advantage of our encoding to enable the bidirectional transfer of analysis and verification techniques between $\pi$ and lcc. First, we formalize the transfer of behavioral equivalences from lcc to $\pi$. Second, by coupling our encoding with a session type system derived from a linear logic interpretation of session types [CP10, CPT15], we identify a class of deadlock-free lcc processes which, to our knowledge, has not been identified before.

- Our third contribution, addressing (b) above, is twofold. First, we introduce lcc+, an extension of lcc having abstractions with local information. Second, building upon the approach in [HL09], we endow lcc+ processes with a type system. With these we are able to precisely stipulate which variables can be abstracted, thus having a way to limit the generality of abstractions; in turn, this enables us to faithfully represent hiding and scope extrusion in $\pi$.

- Our fourth contribution concerns $\pi+$, the extension of $\pi$ with constructs for session establishment. Unlike $\pi$, $\pi+$ allows to explicitly account for the two phases in session-based concurrency mentioned above. We encode $\pi+$ into lcc+. By exploiting the power of declarative specifications, we implement the session establishment phase by embedding a well-known authentication protocol (the Needham-Schroeder-Lowe protocol [Low96]). As such, this encoding highlights the convenience of the extended abstractions and of the type system for lcc+; it is shown to satisfy the same correctness properties as the first encoding.

**Organization.** This paper is structured as follows: §2 describes session-based concurrency and introduces the key ideas in our approach. §3 introduces $\pi$ and lcc. The encoding of $\pi$ in lcc is given in §4. We discuss the bidirectional transfer of techniques between $\pi$ and lcc in §5. The definition of lcc+ (lcc with linear abstractions with local information) and its type system are presented in §6. Our encoding from $\pi+$ processes into (well-typed) lcc+ processes is detailed in §7. We close by discussing related work (§8) and collecting some concluding remarks (§9). The appendices contain omitted proofs.
This paper is an extended, revised version of the conference paper [CRLP15], which appeared in the Proceedings of PPDP’15. With respect to [CRLP15], the results in §5 on the bidirectional transfer of reasoning techniques between $s\pi$ and $1cc$ exploiting the first encoding, are completely new. Also, here we formalize in detail the relationship between $s\pi$ and $s\pi^+$. Moreover, in this presentation we complement [CRLP15] by providing full technical details, additional examples, and extended comparisons with related work.

2 Overview of Key Ideas

In this section we informally illustrate our approach and main results. In our examples, we use $s\pi$ and $1cc$ processes, whose precise syntax and semantics will be introduced in the following section.

Session-Based Concurrency in a Basic Scenario. Consider the following simple protocol between a client and an online store:

1. The client sends to the store a description of an item that she wishes to buy.
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, the client may choose to purchase the item or to end the transaction.

Using session types [HV98], this protocol may be expressed as follows. From the client’s perspective, the session type

\[ S = !\text{item}. ?\text{price}. \oplus \{ \text{buy} : !\text{ccard}. ?\text{invoice}. \text{end}, \text{quit} : !\text{bye}. \text{end} \} \]

says that the output of a value of type $\text{item}$ (denoted $!\text{item}$) should be followed by the input of a value of type $\text{price}$ (denoted $?\text{price}$). These two actions should precede the selection (internal choice, denoted $\oplus$) between two different behaviors distinguished by labels $\text{buy}$ and $\text{quit}$: in the first behavior, the client sends a value of type $\text{ccard}$, then receives a value of type $\text{invoice}$, and then closes the protocol ($\text{end}$ denotes the concluded protocol); in the second behavior, the client emits a value of type $\text{bye}$ and closes the session.

From the store’s perspective, we would expect a protocol that is complementary to $S$. We would have:

\[ T = ?\text{item}. !\text{price}. \& \{ \text{buy} : ?\text{ccard}. !\text{invoice}. \text{end}, \text{quit} : ?\text{bye}. \text{end} \} \]

Indeed, after receiving a value of type $\text{item}$, the store sends a value of type $\text{price}$ back to the client. Subsequently, the store offers (external choice, denoted $\&$) two behaviors to the client, identified by labels $\text{buy}$ and $\text{quit}$. All the behaviors in $T$ match those in $S$, i.e., when the client sends, the store receives, and vice versa; when the client selects, the store offers a choice, and vice versa. Formally, the expected complementarity between types such as $S$ and $T$ is given by session type duality (see, e.g., [BDGK14]). The intent is that implementations derived from dual session types will respect their ascribed (complementary) protocols at run-time, avoiding communication mismatches and other insidious errors.

To illustrate the relationship between session types and $s\pi$ processes, we introduce some notation. We write $x. v. P$ and $x(y). P$ to denote output and input along name $x$, with continuation $P$. Also, given a finite set $I$, we write $x \triangleright \{ \text{lab}_i : P_i \}_{i \in I}$ to denote the offer of labeled processes $P_1, P_2, \ldots$, along name $x$, and $x \triangleleft \text{lab}. P$ to denote the selection of a process label lab along $x$. Moreover, process $b? (P) : (Q)$ denotes
a conditional expression which executes $P$ or $Q$ depending on boolean $b$. Process $P_x$ below is a possible implementation of $S$ along $x$:

$$P_x = \pi \text{book}. x(z). (z \leq 20)? (x \triangleleft \text{buy}. \pi 5406. x(inv).0) : (x \triangleleft \text{quit}. \pi \text{end}.0)$$

In $P_x$, the decision of which option offered by the store is chosen is implemented by a conditional process: the purchase will take place only if the item (a book) is within a $20 budget. The relationship between session types (such as $S$) and operational processes in $\pi$ (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 (see, e.g., the recent survey [HLV+16]). In particular, the key rôle that linearity and linear logic play in session type systems has been clarified in [CPT01Wad12].

The Basic Scenario with Declarative Conditions. $\pi$ processes appropriately describe the communication actions of protocols (which are typically well-defined), but can be less adequate to express the (contextual) conditions that influence partners and their interactions, which are usually hard to know, predict and/or control. Consider now a variation of the above protocol, in which the last step is specified as follows:

3’. Depending on both the item’s price and on the occurrence of event $e$ (say, a temporarily available discount), the client may choose to purchase the item or to end the transaction.

Events (and event detection) are not unusual features in structured protocols; they may add flexibility to specifications by describing the behavior of a communicating system with respect to external sources (such as, e.g., non-communicating components and the communication infrastructure). The presence of a certain event may, e.g., dictate the moment when a communication protocol should be initiated. For instance, in the context of smart grids [LH17], once an abnormal fluctuation on the electrical load of a powerplant is detected, a load-balancing protocol including other power sources should be initiated.

As illustrated by the above alternative protocol step, even though events do not necessarily represent a communicating action, they may influence the structured communication behavior between partners. The influence of event-based behaviors in languages and reasoning techniques for session-based concurrency has been studied in, e.g., [KYH11, DP16].

Events and event-based requirements are just but instances of declarative requirements, which appear difficult to express in the (session) $\pi$-calculus. In our view, the (session) $\pi$-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). Indeed, an appropriate formalization of Step 3’ above in the standard session $\pi$-calculus is far from trivial, because one would unavoidably need to assimilate events to synchronization on names — a far-fetched relationship, especially if the event $e$ does not directly concern communication actions.

Our Approach. We are thus interested in formalisms in which operational and declarative requirements can be jointly specified. We focus on declarative process models based on concurrent constraint programming (ccp) [Sar93]. (See also [BM08b, GPV10, ORV13] for surveys on ccp.) Our goal is to use lcc [FRS01Hae11], a ccp language with linearity, as a unified basis for both operational and declarative requirements in session-based concurrency.

In ccp, processes interact via a global store by means of tell and ask operations. Processes may add constraints (pieces of partial information) to the store by means of tell operations; using ask operations
processes may query the store about some constraint and either execute a process or suspend, depending on the result of the query. These queries on the store are governed by a constraint system, a parametric structure that specifies entailment relationships between constraints. The store thus defines a synchronization mechanism, and both communication-based and external events can be modeled as constraints in the global store. In CCP languages such as 1cc and utcc [OV08a], ask operations are powerful enough to represent mobility of names as in the π-calculus. That is, the key operational mechanisms of the (session) π-calculus (communication of names and scope extrusion) admit (low-level) representations as declarative processes in CCP.

In this paper, we study how 1cc can provide a unified basis for specifying and reasoning about session-based concurrency. We introduce several interpretations (encodings) of sπ 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 and reasoning techniques of sπ and 1cc can be transferred across operational and declarative paradigms using our encodings. A common trait in sπ and 1cc is linearity: in sπ, a session type discipline for processes enforces correct, resource-aware session protocols; linearity is also central to 1cc, as we explain next.

Our Encodings: Key Ideas and Applications To illustrate the essence of our encodings of sπ into 1cc, we first informally explain how tell and ask operations work in 1cc. Let c, d and ¯x denote, respectively, constraints and a (possibly empty) vector of variables. While the tell process τ can be seen as the output of c to the store, the linear abstraction ∀x(d → P) may be intuitively read as: if d can be inferred from the current store then P will be executed — that is, the behavior of P depends on the availability of the guard d. Crucially, this inference consumes the abstraction; it may also involve the consumption of constraints in the store and substitution of parameters ¯x in P, cf. §3.2 When ¯x is empty, we write ∀c(d → P).

In sπ, the most elementary computation step is given by the following reduction rule:

\[(νxy)(π v.P | y(z).Q) →_π (νxy)(P | Q{y/z})\]

The rule specifies the synchronization along two complementary session endpoints, represented by the output process π v.P (where v is a basic value or a name) and the input process y(z).Q. In session-based concurrency, no races in communications between endpoints can occur. The fact that x and y denote indeed reciprocal endpoints for the session protocol is specified by the restriction operator (νxy), which binds them both. All reductions in sπ occur enclosed by a restriction context.

Our first encoding, denoted \( [\cdot]_{sπ} \), translates well-typed sπ processes into 1cc processes (cf. §4). The essence of this declarative interpretation is already manifest in the translation of output- and input-prefixed processes:

\[ [π v.P]_{sπ} = \text{snd}(x, v) \parallel ∀z(\text{rcv}(z, v) \otimes \{x:z\} \rightarrow [P]_{sπ}) \]
\[ [x(y).Q]_{sπ} = ∀y, w(\text{snd}(w, y) \otimes \{w:x\} \rightarrow \text{rcv}(x, y) \parallel [Q]_{sπ}) \]

We use predicates snd(x, v) and rcv(x, y) to model synchronous communication in sπ; we use constraint \( \{x:z\} \) to indicate that x and z are 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 behavior of source sπ process (cf. Thm.4.12). The encoding of \( x \triangleright \{\text{lab}_i : P_i\}_{i \in I} \) and \( x \triangleleft \text{lab} \). P follows similar lines. This interpretation already attests the expressivity of linear abstractions in representing name passing and scope extrusion in sπ.

The encoding \( [\cdot]_{sπ} \) (and its associated correctness properties) enables us to give 1cc specifications which combine representations of operational constructs in sπ and declarative processes that rely on partial
information based on constraints. We may, for instance, “plug” declarative interpretations of operational constructs into larger (declarative) specifications that include behaviors hard to model precisely in $\pi$.

As an example of such a combination, consider again the $\pi$ process given in [1]. We can now give a very simple formalization of Step 3’ above. Using our encoding $[\cdot]_{\pi}$ we could first define the $\lambda \! \! \varepsilon$ process

$$R_z = \forall e (e \otimes (z \leq 20) \rightarrow [x \bowtie \text{buy}. \pi 5406. x(\text{inv}).0]_{\pi}) \parallel [\forall e (e \otimes (z > 20) \rightarrow [x \bowtie \text{quit}. \pi \text{end}.0]_{\pi})$$

which uses the linear multiplicative conjunction $\otimes$ to include the presence of event $e$ into the decision of buying the item or quitting the protocol. Indeed, $\lambda \! \! \varepsilon$ languages treat constraints as logic formulas; in $\lambda \! \! \varepsilon$, such formulas come from Girard’s linear logic $[\text{Gir87}]$. In $R_z$ above, processes $[x \bowtie \text{buy}. \pi 5406. x(\text{inv}).0]_{\pi}$ and $[x \bowtie \text{quit}. \pi \text{end}.0]_{\pi}$ are “macros” that correctly capture the essence of the expected operational session behavior; they are embedded in a (declarative) process context that captures the event-based essence of the requirement in a natural way.

Process $R_z$, however, only corresponds to part of the behavior in [1]. It is easy to consider an $\pi$ process context (a process with a hole, denoted ‘[–]’ below), and to insert $R_z$ into the $\lambda \! \! \varepsilon$ context $C[–]$ resulting from encoding such a context. More precisely, we have:

$$C[–] = [\pi \text{book}. x(z).[–]]_{\pi}$$

$$= \text{snd}(x, \text{book}) \parallel \forall w (\text{rcv}(w, \text{book}) \otimes \{x:w\} \rightarrow \forall z, w (\text{snd}(w, z) \otimes \{w:x\} \rightarrow \text{rcv}(x, z) \parallel ([–])_{\pi}))$$

That is, the $\lambda \! \! \varepsilon$ context $C[–]$ corresponds to the part of $P_z$ in [1] that precedes the labeled selection: it is obtained simply by expanding our encoding $[\cdot]_{\pi}$. Then, process $C[R_z]$ would correspond to the declarative interpretation of $P_z$, enriched with event-based behavior — notice that the free parameter $z$ in $R_z$ (the price of the requested item) is now bound by the abstraction in by $C[–]$. As we will see, the compositional character of our encodings will be crucial to enable process specifications featuring non-trivial combinations of operational macros (such as $[x \bowtie \text{quit}. \pi \text{end}.0]_{\pi}$) with declarative constructs (such as event-based processes). For instance, one could further enrich $C[–]$ by embedding other event-based constructs representing additional declarative requirements.

**Transfer of Reasoning Techniques** As mentioned above, a key motivation to providing declarative interpretations of $\pi$ processes is the possibility of transferring reasoning techniques between operational and declarative models.

In §5 we explore two ways of transferring analysis and verification techniques between $\pi$ and $\lambda \! \! \varepsilon$. First, given that notions of observational equivalence have not been yet developed for $\pi$ processes, we define a behavioral equivalence by coupling our encoding $[\cdot]_{\pi}$ and the observational equivalence for $\lambda \! \! \varepsilon$ processes defined in [Hae11]. Second, we identify a class of deadlock-free $\lambda \! \! \varepsilon$ processes by adapting our encoding $[\cdot]_{\pi}$ to translate a class of session $\pi$-calculus processes that, unlike well-typed $\pi$ processes, enjoys deadlock-freedom directly by typing. We illustrate these two developments by means of examples.

Going back to our running example, consider a variant of the store that simultaneously serves multiple client requests (in $\pi$, this variant could be defined as, e.g., a persistent recursive server process). Consider also a client that performs two different requests to the (persistent) online store; these two requests are treated as different session protocols. The following processes are two different implementations for this client:

$$B_1 = \overline{t_1} \text{item1}. \overline{t_2} \text{item2}. b_1(w_1). b_2(w_2). (F_{b_1} \parallel F_{b_2})$$

$$B_2 = \overline{t_1} \text{item1}. b_1(w_1). F_{b_1} \parallel \overline{t_2} \text{item2}. b_2(w_2). F_{b_2}$$
The different sessions are carried out along two different names, $b_1$ and $b_2$. Above, we write $F_{b_1}$ and $F_{b_2}$ to stand for the processes implementing the purchasing routine in each session. Clearly, processes $B_1$ and $B_2$ represent different options for interleaving the behavior of the two (separate) session behaviors. In an untyped setting, no reasonable notion of observational equivalence should decree $B_1$ and $B_2$ as being behaviourally equivalent — intuitively speaking, $B_2$ is “more parallel” than $B_1$ and has more immediate visible actions. However, once we take the session protocol information into account, it is immediate to see that $b_1$ and $b_2$ implement two independent protocols, and that $B_1$ and $B_2$ should be considered as behaviorally equivalent. To formalize this, in §5.1 we use our encoding $\llbracket \cdot \rrbracket_\pi$ to define an equivalence $\approx_\pi$ on $\pi$ processes in terms of a behavioral equivalence $\approx_1$ defined for $\text{lcc}$ processes. Roughly speaking, we will say that $B_1 \approx_\pi B_2$ if $\llbracket B_1 \rrbracket_\pi \approx_1 \llbracket B_2 \rrbracket_\pi$ under $\text{lcc}$ process contexts that represent the different session protocols implemented in $B_1$ and $B_2$.

In §5.2, we turn our attention to deadlock-freedom, an important liveness property in both operational and declarative settings. Deadlocks are an important class of errors in communicating programs; their presence typically means that some components may not be able to finish their execution due to unfulfilled communication requirements. Consider, for instance, the following $\pi$ process — the archetypical example of a deadlocked session process:

$$R = (\nu x_1 x_2)(x_1(y), x_2(z)) \cdot v_2.0 | x_2(z), x_1 v_1.0)$$

Process $R$ will never evolve: the input in $x_1$ on the subprocess on the right can only be resolved if the subprocess on the left has previously communicated over $x_2$, but such process requires the synchronization with the continuation of the process of the right, generating a circular dependency. Deadlocks are a particularly insidious class of errors for session $\pi$-calculi processes. Since standard type systems for session $\pi$-calculi do not exclude deadlocks (indeed, the type system for $\pi$ in [Vas12] admits deadlocked processes as well-typed), a number of sophisticated typing systems that statically rule out deadlocks have been proposed; see, e.g., [DDY07, CD10, CDM14, CDYP16].

Deadlocks are also a problematic issue for ccp languages, where (global) suspension is a more common synonym [CFM94]. Recall that ask operations are defined in terms of guards, whose entailment triggers the execution of a process; if no guard in a ccp process is enabled, then the process suspends until some other process adds information to the store that entails the guard. Global suspension is the situation in which all components of a parallel process suspend. It is easy to reproduce the circular dependency in $R$ in terms of (nested) tell and ask operations in $\text{lcc}$. Indeed, since our encoding $\llbracket \cdot \rrbracket_\pi$ preserves the structure/nesting of prefixes in $\pi$, the $\text{lcc}$ process $\llbracket R \rrbracket_\pi$ will be globally suspended. To our knowledge, however, $\text{lcc}$ still lacks techniques for (global) suspension analysis, required to exclude processes such as $\llbracket R \rrbracket_\pi$.

We develop a technique for transferring deadlock-freedom guarantees from session $\pi$-calculi to $\text{lcc}$. Rather than $\pi$, our starting point is the typed framework developed in [CPT10, CPT15], where session types are interpreted as linear logic propositions, in the style of the Curry-Howard correspondence. Due to these deep logical foundations, well-typed processes in $\text{lcc}$ respect their session protocols (as in [Vas12]) and are deadlock-free (unlike [Vas12]). Indeed, processes such as $R$ above are not typable in $\text{CPT10, CPT15}$. We adapt the encoding $\llbracket \cdot \rrbracket_\pi$ to this framework and establish its correctness. The adapted encoding, denoted $\llbracket \cdot \rrbracket_\pi^+$, suggests a new class of $\text{lcc}$ processes which are deadlock-free.

**Generalized Abstractions and $\text{lcc}^+$**. In the final part of the paper, we address the overly powerful expressiveness that linear abstractions in $\text{lcc}$ may have in representing certain settings. Indeed, linear abstractions in $\text{lcc}$ may express forms of scope extrusion not possible for untyped $\pi$ processes; see §6.2 for a detailed example of this anomaly. We consider $\text{lcc}^+$, a variant of $\text{lcc}$ with linear abstractions with
These extended abstractions are denoted as follows:

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

In the refined construct, $d$ is a piece of local information (i.e., not publicly available in the store) used jointly with the global information $e$ to trigger process $P$. As a simple example, $d$ could represent a session key that is known to some, but not all, participants in a protocol. Abstractions with local information strictly generalize linear abstractions in $\text{lcc}$, which act on the global store. Indeed, writing $1$ to denote logical truth, the abstraction $\forall \vec{x}(e \to P)$ in $\text{lcc}$ corresponds to $\forall \vec{x}(1; e \to P)$ in $\text{lcc}^+$.

Although helpful, local information alone does not suffice to properly limit the power of abstractions: we need to disallow abstracting on variables that should remain local to the session. To this end, we introduce a type system on $\text{lcc}$ processes that controls variables in abstractions (§6.3). By distinguishing between restricted and unrestricted variables in predicates, we shall say that a $\text{lcc}^+$ process $P$ is well-typed if for every abstraction $\forall \vec{x}(d; e \to Q)$ in $P$, variables $\vec{x}$ are unrestricted.

In §7 we illustrate the expressive power of $\text{lcc}^+$. We develop an encoding of $\pi^+$, the extension of $\pi$ with constructs for session establishment, into $\text{lcc}^+$. Roughly speaking, in $\pi^+$ the execution of a protocol between two parties is preceded by the (secure) exchange of an explicit session key. In the encoding developed in §7, denoted $\llbracket \cdot \rrbracket_\pi^+$ (where mapping $f$ records pairs of co-variables), the session key (used by the two end-points) is treated as local information in the encoding of protocol synchronizations. The encoding $\llbracket \cdot \rrbracket_\pi^+$ relies on a security constraint system $\text{[OV08a]}$. Let $\text{ch}(x; e)$ denote a constraint asserting that the restricted variable $x$ is a channel ($e$ says that the predicate involves no unrestricted variables). In the presence of local information, output and input processes are represented roughly as follows:

$$\llbracket \pi v.P \rrbracket_\pi^+ = \text{snd}(x; v) || \forall e(\text{ch}(x; e); \{x:w\} \otimes \text{ch}(w; e) \otimes \text{rcv}(w, v; e) \to \llbracket P \rrbracket_\pi^+)$$

$$\llbracket w(y).Q \rrbracket_\pi^+ = \forall y(\text{ch}(w; e); \{w:x\} \otimes \text{ch}(x; e) \otimes \text{snd}(w; y) \to \text{rcv}(w, y; e) \to \llbracket Q \rrbracket_\pi^+)$$

By treating constraint $\text{ch}(x; e)$ as local information, we prevent interferences from (malicious) participants aiming at intercepting the identity of session endpoints $x$ and $w$. We show that $\llbracket \cdot \rrbracket_\pi^+$ enjoys operational correspondence (Thm. 7.12) and that the resulting $\text{lcc}^+$ are well-typed (Thm. 7.14). This guarantees that malicious attackers cannot infer private information, such as session keys.

### 3 Preliminaries

We now formally introduce the operational and declarative models that we formally relate in this work: the session $\pi$-calculus (as presented in $\text{[Vas12]}$ – §3.1) and $\text{lcc}$ (as presented in $\text{[Hae11, FRS01]}$ – §3.2). Notation $\vec{e}$ denotes a sequence of elements $e_1, \ldots, e_n$ with length $|\vec{e}| = n$.

#### 3.1 The session $\pi$-calculus ($\pi$)

**Syntax.** Assume a countable infinite set of variables $V_\pi$, ranged over by $x, y, \ldots$. We sometimes refer to variables as channel or names. For simplicity, we only consider boolean constants (tt, ff): we use $v, v', \ldots$ to range over variables and constants (values). Also, we use $l, l', \ldots$ to range over labels.

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

$$P, Q ::= \pi v. P \mid x(y). P \mid x \triangleleft l. P \mid x \triangleright \{l_i : P_i\}_{i \in I} \mid \ast x(y). P \mid v? (P) ; (Q) \mid P \parallel Q \mid (\nu x y) P \mid 0$$
Operational Semantics. denoted π identifies processes up to consistent renaming of bound names, denoted that a process performs on its own. It relies on a structural congruence defined as the smallest relation generated by the rules in Fig. 1. Reduction expresses the computation steps then it behaves as which allows us to express infinite behaviors. The conditional ∗ as customary, we assume pairwise distinct labels. Process sometimes refer to x over values of type T. The type syntax includes over qualifiers, \( \nu \) over pretypes, \( \pi \) over types, and \( \Gamma \) denotes contexts.

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

The type syntax includes pretypes and types. Pretype \( !T_1.T_2 \) denotes output, and types a channel that sends a value of type \( T_1 \) and continues according to type \( T_2 \). Dually, pretype \( ?T_1.T_2 \) denotes input, and types a
(Qualifiers) \( q ::= \) 
- \( \text{lin} \) (linear)
- \( \text{un} \) (unrestricted)

(Pretypes) \( p ::= \) 
- \(?T.T\) (receive)
- \(!T.T\) (send)
- \(\oplus\{l_i : T_i\}_{i \in I}\) (select)
- \(\&\{l_i : T_i\}_{i \in I}\) (branching)

(Types) \( T ::= \) 
- \(\text{bool}\) (boolean)
- \(\text{end}\) (termination)
- \(qp\) (qualified pretype)
- \(a\) (type variable)
- \(\mu a.T\) (recursive type)

(Contexts) \( \Gamma, \Delta ::= \) 
- \(\emptyset\) (empty context)
- \(\Gamma, x : T\) (assumption)

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

channel that receives a value of type \(T_1\) and then proceeds according to type \(T_2\). Pretypes \(\oplus\{l_i : T_i\}_{i \in I}\) and \(\&\{l_i : T_i\}_{i \in I}\) denote labeled selection (internal choice) and branching (external choice), respectively.

Types annotate channels, and can be one of the following: (1) \(\text{bool}\), used for constants and variables; (2) \(\text{end}\), which types a channel endpoint that can no longer be used; (3) qualified pretypes, which type the actions executed by a channel; or (4) recursive types for disciplining potentially infinite communication patterns. The approach to recursion taken in \[Vas12\] is equi-recursive; i.e., a recursive type is assumed to be equal to its unfolding. Qualifiers refer to linear or unrestricted behaviors. Intuitively, linearly qualified types are assigned to endpoints occurring in exactly one thread (a process not comprising parallel composition); the unrestricted qualifier allows an endpoint to occur in multiple threads. For each qualifier \(q\), there are predicates \(q(T)\) and \(q(\Gamma)\) defined as follows:

- \(un(T)\) if and only if \(T = \text{bool}\) or \(T = \text{end}\) or \(T = \text{un} p\).  
- \(lin(T)\) if and only if true.
- \(q(\Gamma)\) if and only if \((x : T) \in \Gamma\) implies \(q(T)\).

Session type systems depend on type duality to relate session types with complementary (or 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., \[BDGK14\] for a formal definition. We write \(\bar{T}\) to denote the dual of type \(T\). Typing uses a context splitting operator on contexts, denoted \(\circ\), which maintains the linearity invariant for channels.

**Definition 3.3 (Context splitting).** Let \(\Gamma_1\) and \(\Gamma_2\) be two contexts; we write \(\Gamma_1, \Gamma_2\) to denote their concatenation. The context splitting of \(\Gamma_1, \Gamma_2\), written \(\Gamma_1 \circ \Gamma_2\), is defined as follows:

\[
\begin{align*}
\emptyset \circ \emptyset &= \emptyset \\
\Gamma_1, x : T &= \Gamma_1 \circ \Gamma_2, x : \text{lin}(T) \\
\Gamma_1, x : T &= (\Gamma_1, x : \text{lin}(T)) \circ (\Gamma_2, x : \text{lin}(T)) \\
\Gamma_1, x : T &= (\Gamma_1, x : \text{lin}(T)) \circ \Gamma_2 \\
\Gamma_1, x : T &= \Gamma_1 \circ (\Gamma_2, x : T)
\end{align*}
\]
Theorem 3.4 ([Vas12]). If $\Gamma \vdash_{s\pi} P$ and $P \rightarrow_{\pi} Q$ then $\Gamma \vdash_{s\pi} Q$.

We now collect some results that concern the structure of processes; they all follow [Vas12]. Some auxiliary notions are needed.

Definition 3.5 (Prefixed Processes and Redexes). We say $\pi.v.P$, $x(y).P$, $x \triangleleft l.P$, $x \triangleright \{l_i : P_i\}_{i \in I}$, and $x(y).P$ are processes prefixed at variable $x$. Redexes are processes of one of the following forms: (i) $\pi.v.P | y(z).Q$, (ii) $\pi.v.P | * y(z).Q$, or (iii) $x \triangleleft l_j.P | y \triangleright \{l_i : Q_i\}_{i \in I}$, with $j \in I$.

We then define well-formed processes:
Definition 3.6 (Well-formed process). An \( \pi \) process \( P_0 \) is well-formed if for each of its structural congruent processes

\[
P_0 \equiv_\pi (\nu x_1 y_1) \ldots (\nu x_m y_m)(P \mid Q \mid R) \quad (m \geq 0)
\]

the following conditions hold:

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

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

Notation 3.7 ((Typable) Programs). A process \( P \) such that \( \text{fn}(P) = \emptyset \) is called a program. Therefore, program \( P \) is typable if it is well-typed under the empty environment (\( \vdash_{\pi} P \)).

The following result connects programs and well-formedness:

Lemma 3.8 (Safety [Vas12]). If \( \vdash_{\pi} P \) then \( P \) is well-formed.

We close this section by illustrating typing derivations in the type system proposed in [Vas12].

Example 3.9. The \( \pi \) process

\[
\pi v_1.0 \mid \pi v_2.0 \mid y(\pi).0
\]  

(2)

can be well-typed in the above system under the context \( \Gamma = \{ x : \mu a . \text{un}!\text{bool}.a, y : \text{lin}!\text{bool}.\text{end} \} \). For the sake of presentation assume \( \Gamma' = \{ x : \mu a . \text{un}!\text{bool}.a \} \). The derivation tree is as follows:

\[
\frac{(T:\text{Out}) \vdash_{\pi} \pi v_1.0 \quad (T:\text{Out}) \vdash_{\pi} \pi v_2.0}{(T:\text{Par}) \vdash_{\pi} \pi v_1.0 \mid \pi v_2.0}
\]

\[
\frac{(T:\text{In}) \vdash_{\pi} y(\pi).0}{\vdash_{\pi} \pi v_1.0 \mid \pi v_2.0 \mid y(\pi).0}
\]

where \( D_1, D_2, \) and \( D_3 \) represent corresponding branches of the previous derivation tree, which have been left out for the sake of presentation. We write \( D_1 \) to represent the following derivation:

\[
\frac{(T:\text{Var}) \vdash_{\pi} x : \text{un}!\text{bool}.a \{ S/a \} \quad (T:\text{Nil}) \vdash_{\pi} \pi v_1.0}{\vdash_{\pi} \pi v_1.0}
\]

where \( S = \mu a . \text{un}!\text{bool}.a \). Note that the leftmost premise of the rule: \( \vdash_{\pi} x : \text{un}!\text{bool}.T\{ S/a \} \) can conclude thanks to the equi-recursive treatment of recursive types in [Vas12]. This means that a type \( \mu a . T \) is equivalent to its unfolding. Derivation \( D_2 \) is represented by the following tree:

\[
\frac{(T:\text{Var}) \vdash_{\pi} x : \text{un}!\text{bool}.a \{ S/a \} \quad (T:\text{Nil}) \vdash_{\pi} \pi v_2.0}{\vdash_{\pi} \pi v_2.0}
\]

where \( S = \mu a . \text{un}!\text{bool}.a \). Note that the leftmost premise of the rule: \( \vdash_{\pi} x : \text{un}!\text{bool}.T\{ S/a \} \) can conclude thanks to the equi-recursive treatment of recursive types in [Vas12]. This means that a type \( \mu a . T \) is equivalent to its unfolding. Derivation \( D_3 \) is represented by the following tree:

\[
\frac{(T:\text{Var}) \vdash_{\pi} x : \text{un}!\text{bool}.a \{ S/a \} \quad (T:\text{Nil}) \vdash_{\pi} \pi v_3.0}{\vdash_{\pi} \pi v_3.0}
\]
Lastly, $D_3$ represents the following derivation:

$$
\frac{(T:\text{Var}) \quad \text{un}(\Gamma') \quad \Gamma', y : \text{lin}?\text{bool}.\text{end} \vdash_{\text{ss}} \Gamma', z : \text{lin}?\text{bool}.\text{end}}{(T:\text{In}) \quad \text{un}(\Gamma') \quad \Gamma' \vdash_{\text{ss}} \text{y}(z).0}
$$

Notice that processes may be typed under different environments. For example, the process in \ref{2} could be typed under environment $\Gamma'' = \{ x : \text{mu}.\text{un!bool}.a, y : \text{mu}.\text{un?bool}.a \}$, with a derivation tree similar to the one above. Notice that $\text{mu}.\text{un!bool}.a$ and $\text{mu}.\text{un?bool}.a$ are dual types.

### 3.2 Linear Concurrent Constraint Programming (lcc)

The linear concurrent constraint calculus (lcc) \cite{FRS01,Hae11} is a declarative formalism that enables the specification and analysis of concurrent systems with partial information and linear resources. As in ccp \cite{Sar93}, concurrent processes in lcc interact via a global store that defines a synchronization mechanism for tell and ask operations. lcc has strong ties to linear logic \cite{Gir87} as well as reasoning techniques over processes based on observational equivalences \cite{Hae11} and phase semantics \cite{FRS01}. A distinguishing feature of lcc with respect to other ccp languages is that lcc allows for non-monotonic evolutions of the store, for the ask operator may consume constraints, which are treated as linear resources.

**Syntax.** We assume countably infinite sets $\mathcal{V}_l$, $\Sigma_c$, and $\Sigma_f$ of variables, predicate symbols, and of functions and constants, respectively. First-order terms, built from $\mathcal{V}_l$ and $\Sigma_f$, will be denoted by $t, t', \ldots$. An arbitrary predicate in $\Sigma_c$ is denoted $\gamma(\vec{t})$.

**Definition 3.10 (lcc Syntax).** The syntax for lcc is given by the following grammar:

Constraints $c := 1 \mid 0 \mid \gamma(\vec{t}) \mid c \otimes c \mid \exists \vec{x}.c \mid !c$

Guards $G := \forall \vec{x}(c \to P) \mid G + G$

Processes $P := \tau \mid G \mid P \parallel Q \mid \exists \vec{x}. P \mid !P$

The grammar for constraints defines the pieces of information that can be posted (asked) to (from) the store. Constant 1, the multiplicative identity, denotes truth; constant 0 denotes falsehood. Logic connectives used as constructors include the multiplicative conjunction ($\otimes$), bang ($!$), and the existential quantifier ($\exists \vec{x}$). Notation $c\{\vec{t}/\vec{x}\}$ denotes the constraint obtained by the (capture-avoiding) substitution of the free occurrences of $x_i$ in $c$, with $|\vec{t}| = |\vec{x}|$ and pairwise distinct $x_i$’s. Process substitution $P\{\vec{t}/\vec{x}\}$ is defined analogously.

The syntax for guards includes non-deterministic choices, denoted $G_1 + G_2$, and parametric asks, denoted $\forall \vec{x}(c \to P)$, which spawns process $P\{\vec{t}/\vec{x}\}$ if the current store entails constraint $c\{\vec{t}/\vec{x}\}$; the exact operational semantics for parametric ask operators (and its interplay with linear constraints) is detailed below. When $\vec{x}$ is empty (a parameterless ask), $\forall \vec{x}(c \to P)$ is denoted as $\forall c(c \to P)$.

Besides guards, the syntax of processes includes the tell operator $\tau$ that adds constraint $c$ to the current store. Moreover, process constructs include parallel composition $P \parallel Q$, that has the expected reading, hiding $\exists \vec{x}. P$, which declares $x$ as being local (private) to $P$, and replication $! P$, that provides infinitely many copies of $P$. We use notation $\prod_{1 \leq i \leq n} P_i$ (with $n \geq 1$) to stand for process $P_1 \parallel \cdots \parallel P_n$. Universal quantifiers in ask operators and existential quantifiers in hiding operators bind their respective variables. Given this, the set of free variables in constraints and processes is defined as expected, and denoted $fv(\cdot)$. 


Definition 3.11 (Constraint system). A constraint system is a triple \((C, \Sigma_c, \vdash)\), where \(C\) is a set of constraints given by the grammar above and \(\Sigma_c\) is a signature that contains the predicates \(\gamma(\cdot)\). Relation \(\vdash\) is a subset of \(C \times C\) that defines the non-modal axioms of the constraint system. Relation \(\vdash\) is the least subset of \(C^* \times C\) containing \(\vdash\) and closed by the rules in Fig. 4. We write \(\vdash c \ni d\) whenever both \(\vdash c \cap d\) and \(d \ni c\) hold.

The semantics of \(1cc\) processes is defined as a Labeled Transition System (LTS), that relies on a structural congruence on processes, given next.

Definition 3.12 (Structural Congruence). The structural congruence relation for \(1cc\) processes is the smallest congruence relation \(\equiv_1\) that satisfies \(\alpha\)-renaming of bound variables, commutativity and associativity for parallel composition and summation, together with the following identities:

\[
\begin{align*}
(\text{ScL1}) & \quad P \parallel T \equiv_1 P & (\text{ScL2}) & \quad \exists z. T \equiv_1 T & (\text{ScL3}) & \quad \exists x. \exists y. P \equiv_1 \exists y. \exists x. P & (\text{ScL4}) & \quad !P \equiv_1 P \parallel !_P \\
(\text{ScL5}) & \quad c \otimes d \vdash e & (\text{ScL6}) & \quad P \equiv_1 P' & (\text{ScL7}) & \quad z \notin f(P) & (\text{ScL8}) & \quad P \equiv_1 P' \\
& \quad \overline{c} \parallel d \equiv_1 \overline{c} & & (\text{ScL9}) & \quad P \parallel Q \equiv_1 P' \parallel Q & & (\text{ScL10}) & \quad \exists x. P \equiv_1 \exists x. P' \\
\end{align*}
\]

As customary, a (strong) transition \(P \xrightarrow{\alpha} P'\) denotes the evolution of process \(P\) to \(P'\) by performing the action denoted by the transition label \(\alpha\):

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

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

The LTS for \(1cc\) processes is generated by the rules in Fig. 5. The premise \(mgc(c, \exists x(d \otimes e))\) in Rules (C:Out) and (C:Sync) denotes the most general choice predicate:

Definition 3.13 (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

\[
mgc(c, \exists \vec{y}(d[^{\vec{t}}/\vec{x}] \otimes e))
\]

whenever for any constraint \(e'\), all terms \(\vec{t}\) and all variables \(\vec{y}\), if \(c \vdash \exists \vec{y}(d[^{\vec{t}}/\vec{x}] \otimes e')\) and \(\exists \vec{y}e' \vdash \exists \vec{y}e\) hold, then \(\exists \vec{y}(d[^{\vec{t}}/\vec{x}] \otimes e')\) and \(\exists \vec{y}e' \vdash \exists \vec{y}e'\).
(C:In) \[
\frac{c \vdash \exists \vec{x}(d \otimes e)}{\exists \vec{x} \vdash \exists \vec{x}'} \quad \text{mgc} (c, \exists \vec{x}(d \otimes e)) \quad (\vec{x} \cup \vec{x'}) \cap f v (c) = \emptyset \\
\] 

(C:Out) \[
\frac{\exists \vec{x} d \vdash \exists \vec{x}' \exists \vec{x}'' \quad \text{mgc} (c, \exists \vec{x}(d \otimes e))}{\exists \vec{x} \vdash \exists \vec{x}''} \quad (\vec{x} \cup \vec{x}'') \\
\] 

(C:Sync) \[
\frac{c \vdash \exists \vec{x}(d \otimes e) \quad \exists \vec{y} \vdash \exists \vec{y}' \quad \text{mgc} (c, \exists \vec{y}(d \otimes e))}{\exists \vec{y} \vdash \exists \vec{y}'} \quad (\vec{x} \cup \vec{y}) \\
\] 

(C:Comp) \[
\frac{P \sigma \rightarrow \alpha P' \quad \text{ev}(\alpha) \cap f v (Q) = \emptyset}{P \parallel Q \sigma \rightarrow \alpha P' \parallel Q} \\
\frac{P \parallel Q \sigma \rightarrow \alpha P' \parallel Q}{P \parallel G_i \sigma \rightarrow \alpha P' \parallel Q_{-i}} \\
\] 

(C:Ext) \[
\frac{P \sigma \rightarrow \alpha Q \quad \exists y \vdash \exists y' \quad \text{mgc} (c, \exists \vec{x}(d \otimes e))}{\exists y \vdash \exists y'} \\
\frac{\exists y \vdash \exists y' \quad \text{mgc} (c, \exists \vec{x}(d \otimes e))}{P \equiv_1 P' \quad P' \sigma \rightarrow \alpha Q' \quad Q' \equiv_1 Q} \\
\] 

(C:Res) \[
\frac{\exists y \vdash \exists y' \quad \text{mgc} (c, \exists \vec{x}(d \otimes e))}{P \equiv_1 P' \quad P' \sigma \rightarrow \alpha Q' \quad Q' \equiv_1 Q} \\
\frac{P \equiv_1 P' \quad P' \sigma \rightarrow \alpha Q' \quad Q' \equiv_1 Q}{P \equiv_1 Q} \\
\] 

Figure 5: Labeled Transition System (LTS) for \( \text{1cc} \) processes.

Intuitively, the mgc predicate allows us to refer formally to decompositions of a constraint \( c \) (seen as “linear resources”) that do not “lose” or “forget” information in \( c \). This is essential in the presence of linear constraints. For example, assuming that \( c \vdash d \otimes e \) holds, we can see that mgc\((c, d \otimes e)\) holds too, because \( c \) is the precise amount of information necessary to obtain \( d \otimes e \). However, mgc\((c \otimes f, d \otimes e)\) does not hold, since \( c \otimes f \) produces more information than the necessary to obtain \( d \otimes e \).

We briefly comment on the rules of Fig. 5. Rule (C:In) asynchronously receives a constraint; it represents the separation between observing an output and its (asynchronous) reception, which is not directly observable.

Rule (C:Out) formalizes asynchronous tells: using the mgc predicate, the emitted constraint is decomposed in two parts: the first part is actually sent (as recorded in the label); the second part is kept as a continuation. (In the rule, these two parts are denoted as \( d' \) and \( e \), respectively.) Rule (C:Sync) formalizes the synchronization between a tell (i.e., an output) and an ask. The constraint mentioned in the tell is decomposed using the mgc predicate: in this case, here the first part is used (consumed) to “trigger” the processes guarded by the ask, while the second part is the remaining continuation.

Rule (C:Comp) enables the parallel composition of two processes \( P \) and \( Q \), provided that the variables extruded in an action by \( P \) are disjoint from the free variables of \( Q \). Rule (C:Sum) enables non-deterministic choices at the level of guards.

Rules (C:Ext) and (C:Res) formalize hiding: the former rule makes local variables explicit in the transition label; the latter rule avoids the hiding of free variables in the label. Finally, Rule (C:Cong) closes transitions under structural congruence (cf. Def. 3.12).

Given the rules for strong transitions, we will assume the standard notation for the reflexive-transitive closure of \( \text{1cc} \) transitions as \( \rightarrow^* \), to represent zero or more \( \tau \)-labeled transitions. Weak transitions are standardly defined: we write \( P \xrightarrow{\tau} Q \) if and only if \( P \rightarrow^* \tau Q \) and \( P \equiv_1 \rightarrow \tau Q \) if and only if \( P \rightarrow^* \tau \rightarrow_2 P' \rightarrow^* \tau Q' \). We identify \( \tau \)-labeled transitions with reductions: we define \( P \rightarrow_1 Q \) as \( P \equiv_1 \rightarrow_1 \equiv_1 Q \). We write \( \rightarrow_1^k \) to indicate \( k \) consecutive reductions (\( k \geq 1 \)).
Observational Equivalences. To reason about encoding correctness, we shall exploit observational equivalences for lcc processes. We require the following auxiliary definition from [Hae11]:

**Definition 3.14 (D-accessible constraints).** Let \( D \subseteq C \), where \( C \) is the set of all constraints. The observables of an lcc process \( P \) are the set of all \( D \)-accessible constraints defined as follows:

\[
O^D(P) = \{(\exists x. c) \in D \mid \text{there exists } P', P \xrightarrow{\tau} 1 \exists x. (P' \parallel \tau)\}
\]

We now recall the definition of a bisimilarity relation for lcc processes. Let \( D, E \subseteq C \). We say that an action \( \alpha \) is \( DE \)-relevant for a process \( P \) if \( \alpha \) is either a silent action \( \tau \), an input action in \( E \), or an output action \((\vec{x}\vec{c})\vec{e}\) with \( \vec{e} \cap f\vec{e}(P) = \emptyset \) and \( \exists \vec{x}. c \in E \).

**Definition 3.15 (DE-bisimilarity).** Let \( D, E \subseteq C \), where \( C \) is the set of all constraints. A symmetric relation \( R \) on lcc processes is a \( DE \)-bisimulation if for all \( P, Q \) such that \( P \xrightarrow{\alpha} P' \), whenever \( P \xrightarrow{\alpha} P' \) then there exists a \( Q' \) such that \( Q \xrightarrow{\alpha} Q' \) and \( P' \xrightarrow{\alpha} P' \). The largest \( DE \)-bisimulation is called \( DE \)-bisimilarity and is denoted \( \approx_{DE} \).

In the following, we will assume \( D = E = C \); bisimilarity will be denoted by \( \approx_1 \). The intuition behind this assumption relies on the fact that for the constraint systems used in this work, both the “input” constraints \( E \) and “output” constraints \( D \) will be of the form \( \exists x_1, x_2, \ldots, x_n. (\gamma_1(x_1) \otimes \gamma_2(x_2) \otimes \ldots \otimes \gamma_n(x_n)) \) with \( n \geq 1 \), which is also the kind of constraints we can find in \( C \), effectively meaning that \( D = E = C \).

Having introduced \( \pi \) and lcc, we now move on to present our first interpretation of \( \pi \) processes into lcc processes.

## 4 Encoding \( \pi \) in lcc

We will be interested in relating operational and declarative process models by means of encodings, i.e., language translations that enjoy certain encodability criteria. In this section, we propose an encoding of \( \pi \) into lcc. We first present the formal notion of correct translation (encoding) that we consider (§4.1), then we describe the translation of \( \pi \) into lcc (§4.2), and finally establish its correctness (§4.3).

### 4.1 The Notion of Encoding

Our definitions of translations, encodings, and encodability criteria are inspired by those proposed by Gorla for name-passing calculi [Gor10]. Differences with respect to Gorla’s proposal are mainly due to the different computational model of (declarative) lcc processes, which lack point-to-point synchronization based on prefixes. These differences are prominent in the definition of compositional transitions—see below. The notion of encoding builds upon abstract notions of calculi and translations, given next.

**Definition 4.1 (Calculi and Translations).** Assume a countably infinite set of variables \( V \).

- A calculus \( L \) is a tuple \( \langle P, \rightarrow, \approx \rangle \), where \( P \) is a set of processes, \( \rightarrow \) denotes an operational semantics, and \( \approx \) is a behavioral equality on \( P \).

- Given calculi \( L_s = \langle P_s, \rightarrow_s, \approx_s \rangle \) and \( L_t = \langle P_t, \rightarrow_t, \approx_t \rangle \), with sets of variables \( V_s \) and \( V_t \), respectively, a translation from \( L_s \) into \( L_t \) is a pair \( \langle \| \cdot \|, \varphi \rangle \), where \( \| \cdot \| : P_s \rightarrow P_t \) denotes a (process) mapping, and function \( \varphi : V_s \rightarrow V_t^k \) (with \( k \geq 1 \)) is a renaming policy for \( \| \cdot \| \).
Def. 3.11 to define the constraint system used in our translation: $\varphi_{\|}$. We move on to consider operational correspondence parallel composition and restriction. Intuitively, name invariance indicates the behavior of the translation of the translation does not depend on specific substitutions. The definition of $\varphi_{\|}$ accounts for the general case in which single variables in $V_s$ are mapped into $V_t$ as tuples of variables with arity $k$; this will be the case for the encoding that we will present in §7.1.

With a slight abuse of notation, given $\vec{x} = x_1, \ldots, x_n$, we write $\varphi_{\|}(\vec{x})$ to stand for the sequence $(x_1), \ldots, (x_n)$. Also, we may refer to a translation $(\|, \varphi_{\|})$ by referring only to its mapping $\|$. An encoding is a translation that satisfies certain correctness criteria:

**Definition 4.2 (Encoding).** Let $\mathcal{L}_s = (\mathcal{P}_s, \rightarrow_s, \approx_s)$ and $\mathcal{L}_t = (\mathcal{P}_t, \rightarrow_t, \approx_t)$ be calculi in the sense of Def. 4.1. A translation $(\|, \varphi_{\|})$ of $\mathcal{L}_s$ into $\mathcal{L}_t$ is an encoding if it satisfies the following criteria:

1. **Name invariance:** For all $S \in \mathcal{P}_s$ and substitution $\sigma$, we have $\|S\sigma\| = \|S\|\sigma'$, with $\varphi_{\|}(\sigma(x)) = \sigma'(\varphi_{\|}(x))$, for any $x \in V_s$.

2. **Compositionality** (with respect to parallel and restriction): Let $\text{res}_s(\cdot, \cdot)$ and $\text{par}_s(\cdot, \cdot)$ (resp. $\text{res}_t(\cdot, \cdot)$ and $\text{par}_t(\cdot, \cdot)$) denote restriction and parallel composition operators in $\mathcal{P}_s$ (resp. $\mathcal{P}_t$). Then we have:

   \[
   \|\text{res}_s(\vec{x}, P)\| = \text{res}_t(\varphi_{\|}(\vec{x}), \|P\|)
   \]

   \[
   \|\text{par}_s(P, Q)\| = \text{par}_t(\|P\|, \|Q\|)
   \]

3. **Operational correspondence**, i.e., it is sound and complete:

   (a) **Soundness:** For all $S \in \mathcal{P}_s$, if $S \rightarrow_s S'$, there exists $T \in \mathcal{P}_t$ such that $\|S\| \rightarrow_t T$ and $T \approx_t S'$.

   (b) **Completeness:** For all $S \in \mathcal{P}_s$ and $T \in \mathcal{P}_t$, if $\|S\| \rightarrow_t T$, there exist $S'$ such that $S \rightarrow_s S'$ and $T \approx_t S'$.

Intuitively, name invariance ensures that translations respect the declared renaming policy. Compositionality ensures that the translation of a process is defined in terms of the translations of its subprocesses. Differently from [Ger10], since we will consider translations between calculi that differ substantially in their syntactic structure (i.e., $\pi$, $1cc$, and their variants), we define compositionality only in terms of parallel composition and restriction. Operational correspondence is divided in soundness and completeness properties: the former ensures that the behavior of a source process is preserved (up to behavioral equivalence) by the translation in the target calculus; the latter ensures that the behavior of a translated (target) process corresponds to that of some source process.

We will distinguish static and dynamic encodability criteria. Static criteria (name invariance and compositionality) refer to structural properties of the translation; dynamic criteria (operational correspondence) relate the behavior of a target process and that of its corresponding source process.

### 4.2 Translating $\pi$ into $1cc$

We move on to consider $\pi$ and $1cc$ as source and target calculi in a translation. We first instantiate Def. 3.11 to define the constraint system used in our translation:
Fig. 6: Session constraint system: Predicates.

\[ \Sigma \overset{\text{def}}{=} \text{rcv}(x, y) \mid \text{snd}(x, y) \mid \text{sel}(x, l) \mid \text{bra}(x, l) \mid \{x:y\} \]

\[
\begin{align*}
\llbracket \pi . P \rrbracket_{\Sigma} &= \text{snd}(x, v) \parallel \forall z (\text{rcv}(z, v) \otimes \{x:z\} \rightarrow \llbracket P \rrbracket_{\Sigma}) & z \notin f_v(P) \\
\llbracket x(y). P \rrbracket_{\Sigma} &= \forall y, w(\text{snd}(w, y) \otimes \{w:x\} \rightarrow \text{rcv}(x, y) \parallel \llbracket P \rrbracket_{\Sigma}) & w, z \notin f_v(P) \\
\llbracket x \leftarrow l. P \rrbracket_{\Sigma} &= \text{sel}(x, l) \parallel \forall z (\text{bra}(z, l) \otimes \{x:z\} \rightarrow \llbracket P \rrbracket_{\Sigma}) & z \notin f_v(P) \\
\llbracket x \triangleright \{l_i; P_i\}_{i \in I} \rrbracket_{\Sigma} &= \forall l, w(\text{sel}(w, l) \otimes \{w:x\} \rightarrow \text{bra}(x, l) \parallel \prod_{1 \leq i \leq n} \forall \epsilon (l = l_i \rightarrow \llbracket P_i \rrbracket_{\Sigma})) & w, z \notin f_v(P) \\
\llbracket v? (P) : (Q) \rrbracket_{\Sigma} &= \forall \epsilon (v = 1 \rightarrow \llbracket P \rrbracket_{\Sigma}) \parallel \forall \epsilon (v = 0 \rightarrow \llbracket Q \rrbracket_{\Sigma}) \\
\llbracket (\nu x y) P \rrbracket_{\Sigma} &= \exists x, y, (!{x:y}) \parallel \llbracket P \rrbracket_{\Sigma} \\
\llbracket \ast x(y). P \rrbracket_{\Sigma} &= ! \llbracket x(y). P \rrbracket_{\Sigma} \\
\llbracket P | Q \rrbracket_{\Sigma} &= \llbracket P \rrbracket_{\Sigma} \parallel \llbracket Q \rrbracket_{\Sigma} \\
\llbracket 0 \rrbracket_{\Sigma} &= \top
\end{align*}
\]

Fig. 7: Translation from \(s\pi\) to \(1cc\).

**Definition 4.3 (Session Constraint System).** Let \(\Sigma\) be the predicates given in Fig. 6. The session constraint system is the tuple \((C, \Sigma, \cup_C)\), where \(C\) is the set of all constraints obtained by using linear logic operators \(!, \otimes\) and \(\exists\) over the predicates of \(\Sigma\) and \(\cup_C\) is given by the rules in Fig. 4.

In Fig. 6, predicate \(\text{rcv}(x, y)\) denotes the acknowledgement of an action that receives through channel \(x\) a value denoted by \(y\); conversely, predicate \(\text{snd}(x, y)\) represents the acknowledgement of an action that sends through channel \(x\) a value denoted by \(y\). Predicates \(\text{sel}(x, l)\) and \(\text{bra}(x, l)\) represent the acknowledgement of actions that send and receive labels through channel \(x\), respectively. We write \(\{x:y\}\) to denote a predicate that connects co-variables \(x\) and \(y\).

We may now formally introduce the translation:

**Definition 4.4 (Translation of \(s\pi\) into \(1cc\)).** We define the translation from \(s\pi\) programs into \(1cc\) processes as the pair \((\llbracket \_ \rrbracket_{s\pi}, \varphi_{s\pi})\), where:

(a) \(\llbracket \_ \rrbracket_{s\pi}\) is the process mapping defined in Fig. 7

(b) \(\varphi_{s\pi}(x) = x\), i.e., the identity function.

Some intuitions on the mapping \(\llbracket \_ \rrbracket_{s\pi}\) in Fig. 7 follow. A first difference between \(s\pi\) and \(1cc\) concerns (a)synchronous communication: since \(s\pi\) is synchronous and \(1cc\) is asynchronous, we exploit the constraints in Fig. 6 as acknowledgment messages to ensure appropriate synchronizations.

To represent an input-output synchronization in \(s\pi\), our translation first synchronizes using message \(\text{snd}(\_\_\_),\) which is posted by the translation of output and mentions both the communication subject and object (i.e., the source name and the intended value). The translation of input may then consume this message to obtain both subject and object using a linear abstraction; it also checks that a co-variable restriction
holds: this is to enforce synchronization between intended source endpoints. Subsequently, the translation emits a message \( \text{rcv}(\cdot, \cdot) \) and spawns its continuation; upon reception/consumption of this message, the translation of output may spawn its own continuation. In this scheme, parameterized ask operators play an important rôle, as they induce substitutions. The translation of a branching-select synchronization follows a similar strategy, using \( \text{bra}(\cdot) \) and \( \text{sel}(\cdot) \) as acknowledgment messages. In this case, the exchanged value is one of the pairwise distinct labels; depending on the received label, the translation of branching will spawn exactly one continuation, as expected.

The translation of conditional expressions makes both branches available for execution; we use a parameterized ask to ensure that only one of them will be executed. The translation of restriction provides infinitely many copies of the appropriate co-variable constraint, using hiding to appropriately regulate the scope of the involved endpoints. The translation of replicated processes simply corresponds to the replication of the input-guarded process. The translations of parallel composition and inaction are self-explanatory.

**Example 4.5.** We present the translation of a very simple \( \pi \) program, featuring an input-output synchronization:

\[
P = (\nu x y)(\pi v.P_1 \mid y(u).P_2) \rightarrow_\pi (\nu x y)(P_1 \mid P_2\{v/u\})
\]

Following the process mapping \([\_]_\pi\), we obtain the \( lcc \) process:

\[
[P]_\pi = \exists x, y. \left( \{\{x:y\} \parallel \left[ \pi v.P_1 \right]_\pi \parallel \left[ y(u).P_2 \right]_\pi \right) \\
= \exists x, y. \left( \{\{x:y\} \parallel \left[ \text{snd}(x, v) \parallel \forall z (\text{rcv}(z, v) \otimes \{z\} \rightarrow \left[ P_1 \right]_\pi) \right) \\
\parallel \forall u, w (\text{snd}(w, u) \otimes \{w:y\} \rightarrow \text{rcv}(y, u) \parallel \left[ P_2 \right]_\pi) \right) \right)
\]

(3)

Process \([P]_\pi\) intuitively behaves as follows. Observe how process \( \text{snd}(x, v) \) (associated to the translation of \([\pi v.P_1]_\pi\)) is meant to interact with the abstraction that results from \([y(u).P_2]_\pi\). The translation of restriction provides unlimited copies of the co-variable constraint \( \{x:y\} \); this suffices to for the translation of input to evolve into process \( \text{rcv}(y, v) \parallel [P_2]_\pi\{v/u\} \). Once that occurs, a similar pattern in the reverse direction is performed: the presence of constraint \( \text{rcv}(y, v) \) as well as the co-variable constraint will release \([P_1]_\pi\), i.e., the translation of continuation of the output. This completes the declarative representation of the reduction from \( P \).

We move on to establishing correctness results for this translation, i.e., to establish that it adheres to the encodability criteria in Def.4.2.

### 4.3 Correctness of the Translation

#### 4.3.1 Static Properties

We show that \([\_]_\pi\) is name invariant with respect to the renaming policy in Def.4.4(b). This proves condition Def.4.2(1):

**Theorem 4.6 (Name invariance for \([\_]_\pi\).** Let \( P \) be a typable \( \pi \) process. Also, let \( \sigma \) and \( x \) be a substitution satisfying the renaming policy for \( [\_]_\pi \) (Def.4.4(b)), and a variable in \( \pi \), respectively. Then \([P\sigma]_\pi = [P]_\pi\sigma'\), with \( \varphi_{[\_]_\pi}(x) = \sigma'(\varphi_{[\_]_\pi}(x)) \) and \( \sigma = \sigma' \).

**Proof.** By structural induction on \( P \). See Appendix A.1 for details. \( \square \)

To simplify the presentation of the semantic encodability criteria, we define *evaluation contexts* for \( \pi \).
Definition 4.7 (Evaluation Contexts ($s\pi$)). The syntax of evaluation contexts in $s\pi$ is given by the following grammar, where $P$ is an $s\pi$ process and `$·' represents a "hole" in said process:

\[ E ::= · | E | P | P | E (\nu xy)(E) \]

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

We now prove compositionality of $[\cdot]_{s\pi}$ with respect to restriction and parallel, as in Def. 4.2 (2):

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

Proof. By case analysis on $E[\cdot]$ and each case by induction on the structure of $P$, using Def. 4.7. See Appendix A.1 for details.

Having established static criteria for $[\cdot]_{s\pi}$, we now investigate operational correspondence, a dynamic encodability criterion.

4.3.2 Operational Correspondence

One important issue to be addressed with $[\cdot]_{s\pi}$ is the non-determinism of $s\pi$. This is crucial, since it is desirable that our translation captures the non-deterministic behavior of processes with unrestricted channels (as in, e.g., a server communicating with multiple clients). This class of processes is not captured in [LOPI0], because it relies on the deterministic language utcc as target calculus. Consider the $s\pi$ program below which, because of its non-determinism, is not encodable in [LOPI0]:

\[ Q = (\nu xy)(\pi v_1.P_1 | \pi v_2.P_2 | y(z).R) \] (4)

Process $Q$ is typable in [Vas12]. In fact, it is quite similar to the process presented in Example 3.9. The main difference is that now there is a restriction establishing $x$ and $y$ as co-variables, and that the continuations are now arbitrary processes $P_1$, $P_2$ and $R$. Assume an environment $\Gamma$ such that $\text{dom}(\Gamma) = \text{fn}(Q)$, with appropriate types for $P_1$, $P_2$ and $R$. We could use Rule (T:Res) to obtain:

\( \Gamma, x : \mu a . \text{un} \, \text{bool} . a, y : \mu a . \text{un} \, ? \text{bool} . a \vdash _{s\pi} \pi v_1.P_1 | \pi v_2.P_2 | y(z).R \)

which then would yield a similar derivation tree to the one presented in Example 3.9.

In terms of possible reductions, we have either:

\[ Q \rightarrow_{\pi} (\nu xy)(P_1 | \pi v_2.P_2 | R(v_1/z)) = Q_1 \quad \text{or} \quad Q \rightarrow_{\pi} (\nu xy)(\pi v_1.P_1 | P_2 | R(v_2/z)) = Q_2 \]

Now consider the utcc process $[Q]_{s\pi}$:

\[ \exists x, y, (\!\{x:y\} \parallel \text{snd}(x, v_1) \parallel \forall z (\text{rcv}(z, v_1) \otimes \{x:z\} \rightarrow [P_1]_{s\pi}) \parallel \text{snd}(x, v_2) \parallel \forall z (\text{rcv}(z, v_2) \otimes \{x:z\} \rightarrow [P_2]_{s\pi}) \parallel \forall z, w (\text{snd}(w, z) \otimes \{w:y\} \rightarrow \text{rcv}(y, z) \parallel [R]_{s\pi}) \}
\]
We write process is the translation for $P$. One can show that it follows directly from Def. 3.6 and from Fig. 7.

Proof. It is instructive to observe the process 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 redexes may constitute a redex (cf. §4.3.1).

Lemma 4.9 (Translated form of a program). Let $P$ be a well-typed $\pi$ program ($\vdash_{\pi} P$) (Not. 3.7), then

$$\llbracket P \rrbracket_{\pi} \equiv_1 \exists x, y. \left( \{x:y\} \parallel \text{snd}(x, v_1) \otimes \text{snd}(x, v_2) \parallel \forall z. (\text{rcv}(z, v_1) \otimes \{x:z\} \rightarrow \llbracket P_1 \rrbracket_{\pi}) \right)$$

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

Proof. Follows directly from Def. 3.6 and from Fig. 7.

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

$$\langle P \rangle_{y_i} = \begin{cases} \forall z. (\text{rcv}(y_i, v) \otimes \{z:y_i\} \rightarrow \llbracket Q \rrbracket_{\pi}) & \text{if } P \equiv_{\pi} (\nu \vec{x} \vec{y})(\pi v. Q | R) \\ \forall z. (\text{bra}(y_i, l) \otimes \{z:y_i\} \rightarrow \llbracket Q \rrbracket_{\pi}) & \text{if } P \equiv_{\pi} (\nu \vec{x} \vec{y})(x_i, a.l.Q | R) \end{cases}$$

We write $\langle P \rangle_{y_i}$ when the co-variable $y_i$ is unimportant.
We may now define:

**Definition 4.11 (Intermediate Processes).** Let $P$ be a typable $\pi$ program, with translated form (cf. Lem. 4.9)

$$\llbracket P \rrbracket_{\pi} = \llbracket C \rrbracket_{\pi} [\llbracket R_1 \rrbracket_{\pi}, \ldots, \llbracket R_i \rrbracket_{\pi}, \ldots, \llbracket R_n \rrbracket_{\pi}]$$

(with $1 \leq i \leq n$).

The set of intermediate processes of $\llbracket P \rrbracket_{\pi}$, denoted $\llbracket \llbracket P \rrbracket_{\pi} \rrbracket$, is defined as

$$\llbracket \llbracket P \rrbracket_{\pi} \rrbracket = \{ S \mid \llbracket P \rrbracket_{\pi} \xrightarrow{\tau} S = \llbracket C \rrbracket_{\pi} [\llbracket R_1 \rrbracket_{\pi}, \ldots, \llbracket R_i \rrbracket_{\pi}, \ldots, \llbracket R_n \rrbracket_{\pi}] \}$$

The previous definitions already give an idea of how $\pi$ reductions are represented by our translation. The translation of a redex must first reach an intermediate lcc process, which 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 transfer of reasoning techniques from lcc to $\pi$:

**Theorem 4.12 (Operational Correspondence for $\llbracket \rrbracket_{\pi}$).** Let $\llbracket \rrbracket_{\pi}$ be the translation in Def. 4.4. 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. $\llbracket P \rrbracket_{\pi} \rightarrow_{\tau} R$, such that $R \approx_{\tau} \llbracket Q \rrbracket_{\pi}$.
   b. (or) $\llbracket P \rrbracket_{\pi} \equiv_{\tau} S' \rightarrow_{\tau} R'$, for some $R', S'$, $R$ such that $R \approx_{\tau} \llbracket Q \rrbracket_{\pi}$.

2. **Completeness:** If $\llbracket P \rrbracket_{\pi} \rightarrow_{\tau} S$ then either:
   a. $P \rightarrow_{\pi} Q$, for some $Q$ and $\llbracket Q \rrbracket_{\pi} \approx_{\tau} S$.
   b. (or) $S \in \llbracket \llbracket P \rrbracket_{\pi} \rrbracket$ and, for some $S'$ and $Q$, we have $S \rightarrow_{\tau} S'$, $P \rightarrow_{\pi} Q$, and $\llbracket Q \rrbracket_{\pi} \approx_{\tau} S'$.

**Proof.** See Appendix A.2.

Informally, cases 1(a) and 2(a) concern the reduction of conditional expressions; cases 1(b) and 2(b) concern other kinds of reductions.

We now state the main result of this section: the translation $\llbracket \rrbracket_{\pi}$ is an encoding, as it satisfies the static and dynamic criteria in Def. 4.2.

**Corollary 4.13.** The translation $\llbracket \rrbracket_{\pi}$ (cf. Def. 4.2) is an encoding.

**Proof.** Direct consequence of Theorems 4.6, 4.8, and 4.12.

## 5 Transfer of Techniques Between $\pi$ and lcc

One key motivation for pursuing declarative interpretations of operational processes is to exploit reasoning techniques available for declarative processes but not for operational ones, and viceversa.

In this section we exploit our encoding $\llbracket \rrbracket_{\pi}$ to transfer properties (behavioral equivalences and type-based liveness analysis) between $\pi$ and lcc. First, we present how to use the observational equivalence in lcc (cf. Def. 3.15) to reason about $\pi$ processes (§5.1). Then, we discuss how a liveness property ensured by a session type system can be transferred to lcc processes (§5.2).
5.1 An Observational Equivalence for $s\pi$

Here we transfer techniques from the declarative setting to the operational one. We focus on one observational equivalence: $\approx_1$ (cf. Def. 3.15), defined for $\lambda\pi$ but not yet developed for $s\pi$.

5.1.1 Definitions

We shall define an observational equivalence for (well-typed) $s\pi$ processes that is defined in terms of $\lambda\pi$ processes resulting from the encoding given in §4.

We require a few auxiliary definitions. The set of characteristic processes contains the "minimal" class of processes that implements a given type.

**Definition 5.1 (Characteristic Process).** Let $p$ be a session pretype (cf. Def. 3.2). Given a channel $x$, the set of characteristic processes of $p$, denoted $\llbracket p \rrbracket^x$, is defined inductively as follows:

$$\llbracket ?T.U \rrbracket^x = \begin{cases} \{x(y).P \mid P \in \llbracket U \rrbracket^x\} & \text{if } T = \text{bool} \\ \{x(y).(P \mid Q) \mid P \in \llbracket T \rrbracket^y, Q \in \llbracket U \rrbracket^x\} & \text{otherwise} \end{cases}$$

$$\llbracket !T.U \rrbracket^x = \begin{cases} \{\exists v.P \mid P \in \llbracket U \rrbracket^x, v \in \{\ttt, \fff\}\} & \text{if } T = \text{bool} \\ \{\exists z.P \mid P \in \llbracket T \rrbracket^z, Q \in \llbracket U \rrbracket^x\} & \text{otherwise} \end{cases}$$

$$\llbracket \oplus \{l_i : T_i\}_{i \in I} \rrbracket^x = \bigcup_{i \in I} \{x \bowtie l_i.P \mid P \in \llbracket U \rrbracket^x\}$$

$$\llbracket \& \{l_i : T_i\}_{i \in I} \rrbracket^x = \{x \triangleright \{l_i : P_i\}_{i \in I} \mid \forall i, P_i \in \llbracket T_i \rrbracket^x\}$$

Given a channel $x$, the set of characteristic processes of a type $T$ is defined inductively as follows:

$$\llbracket q \rrbracket^x = \llbracket p \rrbracket^x$$

$$\llbracket a \rrbracket^x = \llbracket \text{bool} \rrbracket^x = \llbracket \text{end} \rrbracket^x = \{0\}$$

$$\llbracket \mu a.T \rrbracket^x = \llbracket T \rrbracket^x$$

We give some intuitions on Def. 5.1. The cases for input and output session types ($\?T.U$ and $\!T.U$) distinguish whether $T$ is a basic type $\ttt$, $\fff$, or a session type. The latter case is the most interesting, as it involves session passing (delegation). A characteristic process for input receives a session $y$; its continuation is a process $P \mid Q$, where $P$ implements session $T$ along $y$ and $Q$ implements the continuation $U$ along $x$. Dually, a characteristic process for output sends a session of type $T$; its continuation is also a parallel process, where one part implements $U$ and the other implements $T$. That is, the characteristic process delegates (sends) a reference of type $T$ but implements $T$. This ensures proper interactions between the ensuring continuations. The set of characteristic processes for selection (internal choice) $\oplus \{l_i : T_i\}_{i \in I}$ and branching (external choice) $\& \{l_i : T_i\}_{i \in I}$ enjoy a rather direct set-like formulation, in accordance with the set of labels they can select from and offer, respectively.

We can show that processes in the set of characteristic processes of a given type indeed inhabit that type. Before formally stating this property we need some auxiliary results regarding recursive types. We will assume a standard definition for tail-recursive types, which include types such as $\mu a.\!a.a$ and $\mu a.\ttt$. On the other hand, recursive types such as $\mu a.\!a.\text{end}$ will be referred to as object-recursive types.

**Proposition 5.2 (On Inhabited Recursive Types).** For every inhabited recursive type $\mu a.T$, the following holds:

25
(1) \( T = q p \), with \( q \in \{ \text{un}, \text{lin} \} \) (and)
(2) \( T \) can be tail-recursive or object-recursive (and)
(3) if \( \Gamma, x : \mu a.q p \vdash_{\pi} P \) and \( \mu a.q p \) is tail-recursive then \( q = \text{un} \)

Proof. The proof for each numeral can be found in App. B.1.

Notice that although Fig. 2 allows for recursive types such as \( \mu a.\text{end} \) or \( \mu a.\text{bool} \), there are no processes inhabiting such recursive types. The proposition above concerns only recursive types with inhabitants. In particular, Item (1) says that inhabited recursive types can only have the form \( q p \), which indicates that recursive types will pertain only processes on communicating. Item (2) says that recursive types in \( s_{\pi} \) can be tail-recursive (e.g., \( \mu a.!a.\text{bool} \)) or not (e.g., \( \mu a.(!a).\text{end} \)). Lastly, Item (3) indicates that infinite behavior can only come from unrestricted types, since linear types have to be consumed. As an example, it can be easily shown that there is no process inhabiting the (tail-recursive) type \( \mu a.\text{lin ?bool}.a \), as one would require a process that runs infinite inputs, which is not possible in \( s_{\pi} \), as there are no recursion constructs. On the other hand, the (tail-recursive) type \( \mu a.\text{un ?bool}.a \) can type processes such as \( x(y).0 \), since predicate \( \text{un}(\cdot) \) holds for every unrestricted type.

We now proceed to state the main result regarding characteristic processes:

**Proposition 5.3.** Let \( T \) be a type (cf. Fig. 2). For every \( P \in [T]^x \), there exists \( \Gamma \) such that \( \Gamma, x : T \vdash_{\pi} P \).

Proof. The proof proceeds by induction on \( T \); the details can be found on App. B.1.

Given a context \( \Delta \) (cf. Fig. 2), we now define its complementary characteristic process. Intuitively, this is a process that contains a “dual implementation” for each type included in \( \Delta \).

**Definition 5.4 (Complementary characteristic process).** Let \( \Delta = x_1:T_1, \ldots, x_n:T_n \) be a non-empty context, with \( \vec{x} = x_1, \ldots, x_n \). The complementary characteristic process of \( \Delta \) for some sequence \( \vec{y} \) with \( \lvert \vec{y} \rvert = \lvert \vec{x} \rvert \), denoted \( \llbracket \Delta \rrbracket^{\vec{y}} \), is defined as \( Q_1 \mid \ldots \mid Q_n \), where \( Q_i \in [T_i]^x \), for all \( i \in \{1, \ldots, n\} \).

We may now define the equivalence for \( s_{\pi} \) processes derivable via our encoding using the bisimilarity for \( lcc \) (cf. Def. 3.15). For simplicity, and without loss of generality, our definition applies to typed processes in which delegation is treated as a bound output (i.e., no free channel output); this avoids having repeated typing assignments for both the subject and object of a delegated channel.

**Definition 5.5 (Equivalence \( \approx_{\pi} \)).** Let \( P \) and \( Q \) be processes such that \( \Delta, x : T \vdash_{\pi} P \) and \( \Delta, x : T \vdash_{\pi} Q \), with \( \vec{x} = \text{dom}(\Delta) \). Also, let \( \llbracket \Delta \rrbracket^{\vec{y}} \) be the complementary characteristic process of \( \Delta \), with \( \lvert \vec{y} \rvert = \lvert \vec{x} \rvert \) (Def. 5.4). Moreover, let \( [\cdot]_{s_{\pi}} \) be the encoding defined in Fig 2. We write \( P \approx_{\pi} Q \) if and only if

\[
[\nu \vec{x} \vec{y}](P | \llbracket \Delta \rrbracket^{\vec{y}})_{s_{\pi}} \approx_{\pi} [\nu \vec{x} \vec{y}](Q | \llbracket \Delta \rrbracket^{\vec{y}})_{s_{\pi}}
\]

Notice how \( \approx_{\pi} \) focuses on observing a single endpoint \( x \) — the behavior on all the other endpoints in \( P \) and \( Q \) becomes unobservable by the closure implemented by characteristic processes. Intuitively, \( \approx_{\pi} \) allows us to compare implementations for the same session protocols from the perspective of \( x \). We illustrate this equivalence with an example.
5.1.2 Example: Comparing Two Clients

Consider the following situation between two clients and a bookstore:

1. The first and second client open each one a session with the bookstore and send the same book name.
2. The bookstore receives the name from each client and then answers them by providing the price of the book. The bookstore also provides the option to buy the book or to close the session.
3. Depending on the book price, each client decides whether to buy the book or not. Also, the second client can opt to introduce a discount code on the price of the book.

In this example, we may see that since there are two sessions established then we would need a protocol for each session. In this case each session protocol is essentially the same as the scenario provided in §2. However, we provide a session type for each client endpoint ($b_1$, $b_2$). The types are described as follows:

$$T_1 = \text{lin } \text{!string}. ? \text{int}. \oplus \{ \text{buy} : \text{!string. end}, \text{quit} : \text{!bye. end} \}$$  \hspace{1cm} (6)

$$T_2 = \text{lin } \text{!string}. ? \text{int}. \oplus \{ \text{buy} : \text{!string. end}, \text{code} : \text{!string. ?string. !string. end}, \text{quit} : \text{!bye. end} \}$$  \hspace{1cm} (7)

At this point we would like to provide an $\pi$ process that describes the execution of the protocol. For this example we provide two possible implementations of the process for buyers (we assume processes $F_{b_1}$ and $F_{b_2}$ implement the rest of the protocol):

$$B_1 = b_1 \text{book1} \overline{b_2} \text{book2}. b_1(w_1). b_2(w_2). (F_{b_1} | F_{b_2})$$  \hspace{1cm} (8)

$$B_2 = \overline{b_1} \text{book1}. b_1(w_1). F_{b_1} | \overline{b_2} \text{book2}. b_2(w_2). F_{b_2}$$  \hspace{1cm} (9)

In an untyped setting, $B_1$ and $B_2$ are clearly not behaviorally equivalent: although all visible actions by $B_1$ can be matched by $B_2$, there are actions by $B_2$ that cannot be matched by $B_1$ (for instance, the output on $b_2$). Still, since $b_1$ and $b_2$ implement the same independent protocols one may argue that $B_1$ and $B_2$ are equivalent implementations. Observe both processes are typed by a context $\Delta = \{ b_1 : T_1, b_2 : T_2 \}$.

Formally:

$$\{ b_1 : T_1, b_2 : T_2 \} \vdash_{\pi} B_i \hspace{1cm} (i \in \{ 1, 2 \})$$

This fact enforces the idea that $B_1$ and $B_2$ should indeed behave exactly in the same way. In the following proposition we formalize this idea by using tools transferred from 1cc (Def. 5.5).

**Proposition 5.6.** Let $B_1, B_2$ and $b_2$ be as in (8) and (9). Then $B_1 \approx_{\pi} b_2 \pi B_2$.

**Proof.** The proof proceeds by exhibiting an appropriate bisimulation $R$. See Appendix 5.1 for details. □

Proposition 5.6 focuses on endpoint $b_2$; this choice may seem arbitrary. However, it allows us to focus on only one endpoint—namely, the endpoint that is "blocked" in implementation $B_1$. By focusing on one endpoint we may assess an implementation’s behavior by abstracting from the simultaneous executions of different endpoints. Also, in a sense, this isolation helps to enforce the notion that different sessions should not be able to interfere with each other, unless it is explicitly modeled via name-passing. Observe that if we chose to concentrate on $b_1$ (instead of $b_2$) the two processes would still be equivalent:

**Proposition 5.7.** Let $B_1, B_2$ and $b_1$ be as in (8) and (9). Then $B_1 \approx_{\pi} b_1 \pi B_2$.

**Proof.** The proof proceeds by exhibiting an appropriate bisimulation $R$. The analysis follows the same pattern as the proof of Proposition 5.6. □
It is instructive to assess \(\approx_x^e\) also in terms of the processes that it distinguishes. At a first glance, the requirement of both processes having the same context \(\Delta\) appears to be the difference-defining trait when comparing two processes. However, it turns out that the lcc encoding does provide more information about the output behavior of the two compared processes. In particular, \(\approx_x^e\) is susceptible to values in the sense that it differentiates processes that are typed by the same context but are different in the values they output. Consider, for example, the following process, a slight variant of (9):

\[
B'_2 = \overline{b_1} \text{book}3.b_1(w_1).F_{b_1} \mid \overline{b_2} \text{book}4.b_2(w_2).F_{b_2} \quad (10)
\]

We would like to check whether \(B_1\) and \(B'_2\) are equal or not. We have:

**Proposition 5.8.** Let \(B_1, B'_2\) and \(b_1\) be as in Equations (9) and (10). Then \(B_1 \not\approx_x^{b_1} B'_2\).

**Proof.** Intuitively, there is one action that \(B_1\) cannot simulate from \(B'_2\), and that is outputting the value \text{book}4. In fact, considering the encoding presented in Def. 4.4, one can see that the information contained in the store of the encoded processes would be composed by predicates of the form \((\gamma(x, v))\) where \(\gamma \in \{\text{snd}(\cdot, \cdot), \text{rcv}(\cdot, \cdot), \text{sel}(\cdot, \cdot), \text{bra}(\cdot, \cdot)\}\), \(x\) is a channel and \(v\) is a value.

Taking into account the previous fact and considering that \(\approx_1\) (Def. 3.15) is a labeled bisimulation one can notice that the values \(v\) will be present in the labels used by the transitions. In our example, notice that process \(\llbracket B_1 \rrbracket_{\pi}\) can output a label \(\text{snd}(b_2, \text{book}4)\) which cannot be matched by any possible transition in \(\llbracket B'_2 \rrbracket_{\pi}\). Thus we conclude that \(B_1 \not\approx_{\pi} B'_2\).

The previous considerations lead us to conclude that, being based on lcc semantics, the equivalence \(\approx_x^e\) is sensitive to communicated values. That is, the equivalence is not trivial and goes beyond simple type equality. Thus, when asked to compare the behavior of two different (yet equally typed) \(s\pi\) implementations, we will be inclined to consider \(\approx_x^e\) for all endpoints \(x\). Formally, one could provide a new definition for equality of \(s\pi\) processes:

**Definition 5.9.** Let \(P, Q\) be \(s\pi\) processes such that \(\Gamma \vdash_{s\pi} P\) and \(\Gamma \vdash_{s\pi} Q\). We say that \(P \approx_{s\pi} Q\) holds if and only if \(P \approx_{s\pi} Q\) for every \(x \in \text{dom}(\Gamma)\).

Observe that \(\Gamma\) is finite, and therefore there are finite opportunities for comparing two processes. According to the above definition, we would have \(B_1 \approx_{\pi} B_2\) and \(B_1 \not\approx_{\pi} B'_2\).

### 5.2 A Deadlock-free Fragment of lcc

We now explore how to transfer analysis techniques from session-based calculi to lcc. We will focus on deadlock-freedom, a relevant liveness guarantee for session processes but also for lcc specifications. Informally, deadlock-freedom ensures that processes do not “get stuck” and may interact in a productive way. A simple example of a deadlocked lcc process is the following:

\[
M = \forall y(\text{snd}(x_1, y) \rightarrow \text{rcv}(x_1, y)) \parallel (\text{snd}(x_2, v_2) \parallel \forall \epsilon(\text{rcv}(x_2, v_2) \rightarrow \bot)) \parallel (\forall z(\text{snd}(x_2, z) \rightarrow \text{rcv}(x_2, y) \parallel (\text{snd}(x_1, v_1) \parallel \forall \epsilon(\text{rcv}(x_1, v_1) \rightarrow \bot)))
\]

It is easy to see that \(M\) is globally suspended — it cannot reduce: each (parallel) abstraction is guarding the piece of information that is necessary for the other to advance. This kind of deadlocks are a direct consequence of admitting \(s\pi\) processes as source terms. In fact, process \(M = \llbracket P \rrbracket_{s\pi}\), where

\[
P = x_1(y).\overline{x_2} v_2.0 \mid x_2(z).\overline{x_1} v_1.0 \quad (11)
\]
We would like to have analysis techniques that allow us to rule out processes such as \( M \) and \( P \). Our approach is to identify a class of deadlock-free 1cc processes by exploiting the encoding in Fig. 7 using a class of session-based languages for which deadlock-freedom follows by typing.

Unfortunately, the type system for \( s\pi \) does not guarantee deadlock-freedom—there are typable processes that get stuck, such as, e.g., \( P \) above. Therefore, rather than \( s\pi \), the session-based language of interest in this section will be the session typed framework in \([\text{CPT10, CPT15}]\). Based on a session \( \pi \)-calculus here dubbed \( \pi_{\text{CH}} \), this framework defines an interpretation of session types as linear logic propositions, in the style of the Curry-Howard correspondence. This session type system naturally ensures processes which respect session protocols and are deadlock-free. We will develop an encoding of \( \pi_{\text{CH}} \) into 1cc, and analyze the class of 1cc processes it induces.

In §5.2.1 we introduce the syntax and semantics of \( \pi_{\text{CH}} \), and state the properties for well-typed processes. Then, in §5.2.2 we adapt the encoding in Fig. 7 (from \( s\pi \) to 1cc) to \( \pi_{\text{CH}} \) processes, and describe the transfer of deadlock-freedom from \( \pi_{\text{CH}} \) to 1cc.

### 5.2.1 The \( \pi_{\text{CH}} \) Calculus and Its Logically Motivated Type Discipline

We formally introduce \( \pi_{\text{CH}} \) with the following definitions:

**Definition 5.10 (\( \pi_{\text{CH}} \) Processes \([\text{CPT15, CPT17}]\)).** The syntax for \( \pi_{\text{CH}} \) processes is given by the following grammar:

\[
P, Q ::= \pi v.P \mid x(y).P \mid x < l.P \mid x \triangleright \{ i : P_i \}_{i \in I} \mid P \mid Q \mid 0 \mid (\nu x)P \mid [x \leftrightarrow y] \mid !x(y).P
\]

The syntax of processes is very close to that in Def. 3.1 for \( s\pi \) processes. Main differences concern the restriction operator, which in \( \pi_{\text{CH}} \) only binds one variable: dual endpoints will be implemented by names with the same identity but with complementary behaviors. Also, the syntax of \( \pi_{\text{CH}} \) includes a forwarding process, denoted \([x \leftrightarrow y]\), that intuitively fuses names \( x \) and \( y \) (a sort of explicit substitution). Notions of free/bound names are as expected. The linear logic interpretation of session types based on \( \pi_{\text{CH}} \) relies on bound outputs \((\nu y)\pi y.P\), which are abbreviated as \( (\pi y).P \). This does not represent a loss of expressiveness, since it is known that bound outputs can represent usual free outputs \([\text{Bor98}]\). Instead, a simple representation of the free output process \( \pi v.P \) is given by the process \( \pi(y).([y \leftrightarrow v] \mid P) \).

The operational semantics for \( \pi_{\text{CH}} \) is close to that for \( s\pi \). Here again main differences concern the use and role of the restriction operator (and co-variables) and the treatment of forwarder processes. We require the following definition of structural congruence for \( \pi_{\text{CH}} \) processes:

\[
\begin{align*}
P | 0 =_{\pi_{\text{CH}}} P & \quad P | Q =_{\pi_{\text{CH}}} Q | P & \quad P =_{\pi_{\text{CH}}} Q \text{ if } P =_{\alpha} Q \\
(P | Q) | R =_{\pi_{\text{CH}}} (Q | R) & \quad (\nu x)(\nu z)P =_{\pi_{\text{CH}}} (\nu z)(\nu x)P \\
(\nu x)0 =_{\pi_{\text{CH}}} 0 & \quad (\nu x)P | Q =_{\pi_{\text{CH}}} (\nu x)(P | Q) \quad \text{if } x \not\in \text{fn}(Q) \quad [x \leftrightarrow y] =_{\pi_{\text{CH}}} [y \leftrightarrow x]
\end{align*}
\]

Thus, differences between \( =_{\pi} \) and \( =_{\pi_{\text{CH}}} \) concern the notation for restriction and an extra axiom for forwarders, which formalizes their bidirectionality. We now have:

**Definition 5.11 (Operational Semantics for \( \pi_{\text{CH}} \)).** We define \( \equiv_{\pi_{\text{CH}}} \) as the smallest relation that satisfies the rules in Fig. 3.
Type discipline. The type system in [CPT15] corresponds to propositions in (classical) linear logic:

**Definition 5.12 (πcla Types).** The syntax for πcla types is given by the following grammar:

\[ A, B ::= ⊥ \mid 1 \mid A \otimes B \mid A \otimes B \mid \bigoplus \{i : A_i\}_{i \in I} \mid \& \{i : A_i\}_{i \in I} \mid !A \mid ?A \]

Intuitively, both \(⊥\) and \(1\) are associated to terminated endpoints; it is similar to type \(\text{end}\) for sessions. Type \(A \otimes B\) is assigned to an endpoint that first outputs an object of type \(A\) and then behaves as \(B\); it is therefore related to the pre-type \(!A.B\). Type \(A \otimes B\) is assigned to an endpoint that receives an object of type \(A\) and then proceeds as \(B\); as such, it is related to the pre-type \(?A.B\). Types \(\bigoplus \{i : A_i\}_{i \in I}\) and \(\& \{i : A_i\}_{i \in I}\) represent selection and branching, respectively, with expected readings. Type \(!A\) is assigned to an endpoint able to provide infinitely many copies of behavior \(A\), i.e., a replicated server. Dually, type \(?A\) expresses the capability of requesting copies of behavior \(A\) by synchronizing with an appropriate replicated service.

One advantage of following a linear logic interpretation of sessions is that duality on types is directly informed by linear logic duality:

\[ \top = 1 \quad \bot = 1 \quad \neg A = ?A \quad \neg \neg A = A \quad \bigoplus \{i : A_i\}_{i \in I} = \& \{i : A_i\}_{i \in I} \]

We then say that \(\neg\) represents the dual of \(A\). Following [CPT17], we consider a logical interpretation of session types based a classical linear logic with mix principles, which admit \(1 \rightarrow \bot\) and \(\bot \rightarrow 1\). Therefore, we have \(\bot = 1\), which nicely coincides with the meaning of \(\text{end}\) in session types. In the sequel, we write \(\bullet\) to denote both \(1\) and \(\bot\).

Typing environments are sets of type assignments \(x : A\). There are two environments, denoted \(Δ\) and \(Θ\), which satisfy different structural principles: \(Δ\), the linear environment, is subject only to exchange; \(Θ\), the unrestricted environment, is subject to weakening, exchange, and contraction principles. The empty context is denoted \(\emptyset\). Typing judgments are then of the form \(P \vdash_{πcla} Δ; Θ\). Typing rules are given in Fig. 10 We briefly comment on some of them; see [CPT15] for extended discussions. Rule (T Curt) jointly treats composition and restriction operators: it defines the interaction between two processes which offer dual linear behaviors along the same name \(x\) in the rule. All other behaviors in the processes \(Δ_1\) and \(Δ_2\) in the rule are disjoint: this is essential to avoid the circular dependencies that are at the heart of deadlock processes. Rule (T Curt') enforces composition of shared resources. Rule (T Mix) enables the parallel composition of “independent” processes that do not share linear behaviors. The remaining rules are either self-explanatory or follow their analogues in the type system for \(sπ\) (see Fig. 2).
is the process

Example 5.15. Recall the deadlocked process $P$ given in (1). A variant of $P$ which is well-typed in $\pi_{\text{CH}}$ is the process

$$P' = (\nu x_2)((\nu x_1)(x_1(y),x_2(z)) \cdot 0 | \pi_1(v_1) \cdot 0 | \pi_2(v_2) \cdot 0)$$

which features two independent sessions, $x_1$ and $x_2$, each implementing an input-output synchronization.
Intuitively, all processes that come from the encoding in Def. 5.16 are part of class persistently available equality constraint. This equality allows us to translate as follows:

Although the mappings in Figures 7 and 11 are very similar, some aspects are worth noticing. The translation of \( \mathcal{P} \) processes into \( \mathcal{LCC} \) involves simpler \( \mathcal{LCC} \) processes, since we need not to account for co-variables (as in \( \mathcal{L} \)).

We first define the sub-language of \( \mathcal{LCC} \) processes characterized by the session type system for \( \mathcal{PI} \), and are presented in Appendix B.2.

\[
\begin{align*}
\llbracket v, P \rrbracket_{\mathcal{PI}} &= \text{snd}(x, v) \parallel \forall e(\text{rcv}(x, v) \rightarrow \llbracket P \rrbracket_{\mathcal{PI}}) \\
\llbracket x(y). P \rrbracket_{\mathcal{PI}} &= \forall y(\text{snd}(x, y) \rightarrow \text{rcv}(x, y) \parallel \llbracket P \rrbracket_{\mathcal{PI}}) \\
\llbracket x \ltimes l. P \rrbracket_{\mathcal{PI}} &= \text{sel}(x, l) \parallel \forall e(\text{bra}(x, l) \rightarrow \llbracket P \rrbracket_{\mathcal{PI}}) \\
\llbracket x \triangleright \{ l_i : P_i \}_{i \in I} \rrbracket_{\mathcal{PI}} &= \forall l(\text{sel}(x, l) \rightarrow \text{bra}(x, l) \parallel \prod_{1 \leq i \leq n} \forall e(l_i \rightarrow \llbracket P_i \rrbracket_{\mathcal{PI}}))
\end{align*}
\]

\[
\begin{align*}
\llbracket (\nu x). P \rrbracket_{\mathcal{PI}} &= \exists x. (\llbracket P \rrbracket_{\mathcal{PI}}) \\
\llbracket 0 \rrbracket_{\mathcal{PI}} &= \mathbb{T} \\
\llbracket P \mid Q \rrbracket_{\mathcal{PI}} &= \llbracket P \rrbracket_{\mathcal{PI}} \parallel \llbracket Q \rrbracket_{\mathcal{PI}} \\
\llbracket x \leftrightarrow y \rrbracket_{\mathcal{PI}} &= 1 \llbracket x = y \rrbracket \\
\llbracket !x(y). P \rrbracket_{\mathcal{PI}} &= \top \llbracket x(y). P \rrbracket_{\mathcal{PI}}
\end{align*}
\]

Figure 11: Translation from \( \mathcal{PI} \) to \( \mathcal{LCC} \).

5.2.2 Encoding \( \mathcal{PI} \) in \( \mathcal{LCC} \) and Transfer of Deadlock-freedom

We now translate \( \mathcal{PI} \) into \( \mathcal{LCC} \) by adapting the translation \( \llbracket \cdot \rrbracket_{\mathcal{PI}} \) given in §4.2. Using this adapted translation, we will identify a class of deadlock-free \( \mathcal{LCC} \) processes characterized by the session type system for \( \mathcal{PI} \). The translation of \( \mathcal{PI} \) into \( \mathcal{LCC} \) is defined as follows:

**Definition 5.16 (Translation of \( \mathcal{PI} \) into \( \mathcal{LCC} \)).** We define the translation from \( \mathcal{PI} \) processes into \( \mathcal{LCC} \) processes as the pair \( (\llbracket \cdot \rrbracket_{\mathcal{PI}}, \varphi_{\llbracket \cdot \rrbracket_{\mathcal{PI}}} \rrbracket) \), where:

(a) \( \llbracket \cdot \rrbracket_{\mathcal{PI}} \) is the process mapping defined in Fig. 11

(b) \( \varphi_{\llbracket \cdot \rrbracket_{\mathcal{PI}}} (x) = x \), i.e., the identity function.

Although the mappings in Figures 7 and 11 are very similar, some aspects are worth noticing. The translation \( \llbracket \cdot \rrbracket_{\mathcal{PI}} \) involves simpler \( \mathcal{LCC} \) processes, since we need not to account for co-variables (as in \( \mathcal{L} \)). Also, the translation \( \llbracket \cdot \rrbracket_{\mathcal{PI}} \) considers the forwarding process of \( \mathcal{PI} \) (not present in \( \mathcal{S} \)), which is translated as a persistently available equality constraint. This equality allows us to translate \( \mathcal{PI} \) processes such as:

\[(\nu y)((\nu x)(\pi(z).P \mid [x \leftrightarrow y]) \mid y(w).Q)\]

The translation of \( \mathcal{PI} \) into \( \mathcal{LCC} \) is an encoding, in the sense of Def. 4.2. We omit the associated correctness results; they closely follow the lines described in §4.3 for \( \llbracket \cdot \rrbracket_{\mathcal{S}} \), and are presented in Appendix B.2.

We now move on to establish a deadlock-free class of \( \mathcal{LCC} \) processes exploiting the encoding into \( \mathcal{PI} \). We first define the sub-language of \( \mathcal{LCC} \) processes that is induced by the encoding \( \llbracket \cdot \rrbracket_{\mathcal{PI}} \):

**Definition 5.17 (Class \( \mathcal{LCC}^{\mathcal{PI}} \)).** Given the encoding \( \llbracket \cdot \rrbracket_{\mathcal{PI}} \) in Def. 5.16, the class of processes \( \mathcal{LCC}^{\mathcal{PI}} \) is defined as follows:

\[\mathcal{LCC}^{\mathcal{PI}} = \{ M \mid M \in \mathcal{LCC} \land \exists P \in \mathcal{PI}, M \equiv_{\mathcal{PI}} \llbracket P \rrbracket_{\mathcal{PI}} \}\]

Intuitively, all processes that come from the encoding in Def. 5.16 are part of class \( \mathcal{LCC}^{\mathcal{PI}} \). Thus, processes such as \( P = \text{snd}(x, v) \parallel \forall e(\text{rcv}(x, v) \rightarrow \llbracket P \rrbracket_{\mathcal{PI}}) \) would belong to the \( \mathcal{LCC}^{\mathcal{PI}} \) class. However, processes
such as \( \text{snd}(x, v) \) or \( \forall y (\text{rcv}(x, y) \rightarrow \text{snd}(x, v)) \) are not in \( 1cc^{\pi_\text{C}} \), as there is no corresponding \( \pi_\text{C} \) process that can generate them via \( \llbracket P \rrbracket_{\pi_\text{C}} \).

We define the notion of "live" process for processes in \( 1cc^{\pi_\text{C}} \):

**Definition 5.18 (Live \( 1cc \) processes).** Let \( M \in 1cc^{\pi_\text{C}} \). We write \( \text{live}_{1cc}(M) \) if and only if \( M \equiv_1 \exists \bar{x}. (S \parallel R) \), where \( S \) corresponds to one of the following processes:

1. \( \text{snd}(x, v) \), for some \( x \in \bar{x}, v \).
2. \( \forall y (\text{snd}(x, y) \rightarrow \text{rcv}(x, y) \parallel S) \), for some \( x \in \bar{x}, S \).
3. \( \text{sel}(x, l) \), for some \( x \in \bar{x}, l \).
4. \( \forall l (\text{sel}(x, l) \rightarrow \text{bra}(x, l) \parallel S) \), for some \( x \in \bar{x}, S \).

The following definition encompasses the idea of progress (deadlock-freedom) for \( 1cc^{\pi_\text{C}} \) processes.

**Definition 5.19 (Progress for an \( 1cc^{\pi_\text{C}} \) process).** We say that \( M \in 1cc^{\pi_\text{C}} \) has progress, denoted \( \text{progress}_{1cc}(M) \), if \( \text{live}_{1cc}(M) \) and there exists an \( M' \) such that \( M \overset{\tau}{\rightarrow}_1 M' \).

Intuitively, liveness in this context denotes the potential of executing a \( \tau \)-action; in particular, having one of the necessary pieces for executing such actions (i.e., either putting a constraint in the store or asking for a constraint in it). On the other hand, progress means that the process is actually able to execute a \( \tau \)-action, meaning that all the required ingredients for execution are present.

We connect the notions of liveness for \( 1cc^{\pi_\text{C}} \) (given above) and for \( \pi_\text{C} \) (given in Def. 5.13):

**Lemma 5.20.** Let \( M \in 1cc^{\pi_\text{C}} \) and \( P \) be a \( \pi_\text{C} \) process such that \( \llbracket P \rrbracket_{\pi_\text{C}} = M \). Then \( \text{live}_{1cc}(M) \leftrightarrow \text{live}_{\pi_\text{C}}(P) \).

**Proof.** See Appendix B.2.

The transfer can be formalized as follows:

**Theorem 5.21.** Let \( M \in 1cc^{\pi_\text{C}} \) and let \( P \) be a \( \pi_\text{C} \) process such that \( \llbracket P \rrbracket_{\pi_\text{C}} = M \). If \( P \vdash_{\pi_\text{C}} \emptyset \) and \( \text{live}_{\pi_\text{C}}(P) \) then \( \text{progress}_{1cc}(M) \) holds.

**Proof.** Assume a process \( M \in 1cc^{\pi_\text{C}} \) with \( \llbracket P \rrbracket_{\pi_\text{C}} = M \):

1. \( \text{live}_{\pi_\text{C}}(P) \) (Assumption)
2. \( P \vdash_{\pi_\text{C}} \emptyset \) (Assumption)
3. \( \exists Q, P \rightarrow_{\pi_\text{C}} Q \) (By Thm. 5.14 on (1),(2))
4. \( \text{live}_{1cc}(M) \) (By Lem. 5.20 on (1))
5. \( \exists M', M \rightarrow_1 M' \) (By Thm. B.4 Soundness on (3))
6. \( \text{progress}_{1cc}(M) \) (By Def. 5.19 (4) and (5))

**Theorem 5.21** is not only significant because it establishes a formal route to identify deadlock-free \( 1cc \) processes; it also suggests a class of \( 1cc \) processes, more general than \( 1cc^{\pi_\text{C}} \), for which deadlock-freedom holds by construction. Such a more general class of deadlock-free processes can be spelled out by the following grammar (essentially, a deconstruction of the encoding in Fig. 11):

\[
M ::= p_i(x, v) \parallel \forall e(q_i(x, v) \rightarrow M) \mid \forall y(p_i(x, y) \rightarrow q_i(x, y) \parallel M) \mid M_1 \parallel M_2 \mid ! M \mid \exists x. M \mid \top
\]
where \( p_i, q_i \) denote binary predicates, for simplicity. It is easy to see that the above grammar is a simple
generalization of the syntactic structure of processes in \( \text{lcc}^{\pi \pi} \); indeed, \( \text{lcc}^{\pi \pi} \) corresponds to the instance
in which \( p_1 = \text{out}, q_1 = \text{in}, p_2 = \text{sel}, \) and \( q_2 = \text{br} \).

### 6 An Extension of lcc and Its Type System

We introduce \( \text{lcc}^{+} \), an extension of \( \text{lcc} \) in which abstractions are generalized with local information. We
define a type system for \( \text{lcc}^{+} \) that regulates the power of abstractions, and establish its main properties.

#### 6.1 \( \text{lcc}^{+} \): \( \text{lcc} \) with Linear Abstractions with Local Information

Abstractions in \( \text{lcc} \) 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. This
is the case of session processes after a session has been established. More in general, examples of local
information are (private) keys used in protocols for secure communications.

To remedy this, we consider \( \text{lcc}^{+} \), a variant of \( \text{lcc} \) in which abstractions are generalized so as to ac-
count for local information. The syntax of \( \text{lcc}^{+} \) results from Def\[3,10\] by replacing the abstraction operator
\( \forall \vec{x}(e \rightarrow P) \) (in the grammar for guards) with the following one:

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

The new element is constraint \( d \), a piece of local information used jointly with \( e \) to trigger \( P \). That is, \( d \) is used as additional resource in inferring \( e \) from the global store; still, \( d \) is used locally, for it is not added to the store. This abstraction construct is a generalization, in the sense that \( \forall \vec{x}(e \rightarrow P) \) in \( \text{lcc} \) corresponds to \( \forall \vec{x}(1; e \rightarrow P) \) in \( \text{lcc}^{+} \). The operational semantics of \( \text{lcc}^{+} \) formalizes these intuitions. It is de/f_ined by
the LTS in Fig 5, replacing Rule (C:Sync) with the following rule:

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

\[
\frac{c \otimes d \vdash \exists \vec{y}.(e(\vec{t} / \vec{x}) \otimes f) \quad \vec{y} \cap fv(c, d, e, P) = \emptyset}{\text{mgc} (c \otimes d, \exists \vec{y}.(e(\vec{t} / \vec{x}) \otimes f)) \quad c \otimes d \vdash 0 \Rightarrow c \vdash 0 \quad \tau || \forall \vec{x}(d; e \rightarrow P) \quad \tau \vdash \exists \vec{y}.(P(\vec{t} / \vec{x}) || \vec{f})}
\]

In this new rule, premise \( c \otimes d \vdash 0 \Rightarrow c \vdash 0 \) ensures that only local assumptions which do not conflict
with the information in the global store are allowed. All other notions and definitions for \( \text{lcc} \) processes
will carry over to \( \text{lcc}^{+} \). In the remainder of this paper, we will focus on \( \text{lcc}^{+} \) rather than on \( \text{lcc} \). Next
we define a simple type system for disciplining abstractions in \( \text{lcc}^{+} \). Then, the use of abstractions using
local information will be illustrated in §7.

#### 6.2 A Type System for \( \text{lcc}^{+} \): Motivation

The encoding of \( s\pi \) into \( \text{lcc} \) introduced in §4 relies critically on abstractions to represent synchronizations
in \( s\pi \), as required to encode session communications (including scope extrusions) and their associated
continuations. Unfortunately, the abstraction mechanism in \( \text{lcc} \) is overly powerful for modeling scope extrusion,
in the sense that abstraction can represent scenarios not possible in \( s\pi \) by combining name passing and restriction. Precisely, the private character of synchronizations on restricted channels is not
are conjunctions of predicates applied to terms over the function signature. We consider two environments, \( \Gamma \) and \( \Delta \), which respect the sorting policy by not concerning non-abstractable variables. Intuitively, this means that we distinguish between two sorts of variables: one denoting unrestricted (i.e., privacy-sensitive, non-abstractable) 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 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 \( \nu \), process \( \nu (\pi v. x, y) \, | \, y(z).Q_y \, | \, \nu \) could synchronize according to the session, but could also (wrongly) interact with \( \nu \), thus breaching session privacy. Thus, the (deterministic) reduction in \( 2 \) can no longer be ensured when \( \pi \) processes are compiled down into \( \text{lcc} \) (or \( \text{lcc}^+ \)).

Note that this anomaly is not particular of our encoding \( \llbracket \cdot \rrbracket_{\pi} \); rather, it affects all \( \text{lcc} \) and \( \text{lcc}^+ \) processes that use abstractions to synchronize input-like processes. Scope extensions such as the one possible from the process in \( 3 \) are clearly not possible in \( \pi \); we must limit the power of abstractions so as to preserve the nature and essential assumptions of the restriction operator in \( \pi \). Intuitively, this means that the privacy of session endpoints (inherited from the restriction operator) must be explicitly programmed at the declarative level of \( \text{lcc}^+ \) processes, relying on some extra mechanism to limit abstractions.

To this end, next we develop a simple typing discipline for \( \text{lcc}^+ \), built upon the approach proposed in [HL09] (where the focus is in \( \text{utcc} \) and session-based concurrency is not addressed). Our type system admits only abstractions which adhere to a precisely defined policy for unrestricted and restricted variables. 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 \( \text{lcc}^+ \) abstractions.

A well-typed \( \text{lcc}^+ \) process in our type system is a process in which all abstractions respect the sorting policy by not concerning non-abstractable variables.

The type system is defined in general terms; one application is our encoding of \( \pi \) (the extension of \( \pi \) with constructs for session establishment) into \( \text{lcc}^+ \); see [47]. In this case, the sorting policy applies to the predicates used to represent synchronizations. This way, e.g., we will assume a new signature where \( \text{snd}(x, y) \) is a predicate 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 \( \llbracket (\nu \pi v. x, y) \, | \, y(z).Q_y \rrbracket_{\pi} \) from process \( \llbracket (\nu \pi v. x, y) \, | \, y(z).Q_y \rrbracket_{\pi} \, | \, \nu \) while the former is well-typed, the latter is not (see also Example 6.3).

### 6.3 The Type System

The typing rules for secure patterns/processes are defined in Fig. 12. For simplicity, we assume that patterns are conjunctions of predicates applied to terms over the function signature. We consider two environments,
We consider three kinds of judgments: 

- Judgment $\Delta; \Theta \vdash \cdot c$ 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 $\vdash_{\Lambda} G$

- Finally, a well-typed process $P$ is denoted by $\vdash_{o} P$.

We comment on typing rules in Fig. 12. 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. This rule relies on functions on terms $\text{unr}(t)$, $\text{res}(t)$, and $\text{var}(t)$, yielding, respectively, the set variables appearing unrestricted in $t$ according to the sorting; the set of variables appearing restricted in $t$; and the set of all variables occurring in $t$. Formally, these functions are given by:

- $\text{unr}(x) = \text{res}(x) = \text{var}(x) = \{x\}$ (x is a variable)
- $\text{unr}(\gamma(t_1; t_2)) = \text{unr}(t_2)$
- $\text{res}(\gamma(t_1; t_2)) = \text{res}(t_1) \cup \text{res}(t_2)$
- $\text{var}(\gamma(t_1; t_2)) = \text{var}(t_1) \cup \text{var}(t_2)$

We assume $\text{unr}(x)$, $\text{res}(x)$, and $\text{var}(x)$ extend to vectors $\vec{x}$ in the expected way. Notice that $\text{var}(t) = \text{res}(t) \cup \text{unr}(t)$ but also that $\text{res}(t) \cap \text{unr}(t)$ may be non-empty; in $\gamma(t_1; t_2)$, terms in $t_2$ could contain restricted variables (in nested predicates, for instance, but our encodings do not use nested predicates).
Rule (L:Comβ) identifies the restricted and unrestricted variables in the pattern $c \otimes d$. The set of restricted variables for $c$ must be disjoint from the set of unrestricted variables for $d$, and vice-versa. This avoids treating restricted variables in $c$ or $d$ as unrestricted variables in $c \otimes d$. Typing rules for guards and processes are simple. The most interesting rule is (L:Abs), which says that abstraction $\forall \vec{x} (d; c \rightarrow P)$ is secure as long as variables $\vec{x}$ are unrestricted in the typing for $c$, and no variables in $d$ occur in $\vec{x}$.

The main theorem regarding the type system is type preservation (Thm 6.2), whose proof relies on subject congruence:

**Lemma 6.1.** If $P \equiv_1 Q$ and $\vdash_\circ P$, then $\vdash_\circ Q$.

**Proof.** By induction on the depth of the premise $P \equiv_1 Q$. See App. C

**Theorem 6.2 (Type Preservation).** If $P \overset{\alpha}{\rightarrow}_1 Q$ and $\vdash_\circ P$ then $\vdash_\circ Q$.

**Proof.** The proof proceeds by induction on the depth of the premise $P \overset{\alpha}{\rightarrow}_1 Q$. See App. C

**Example 6.3 (An Ill-typed Process).** As a simple illustration of our type discipline, consider the following process, similar to process $\llbracket x(y).P \rrbracket_{\pi_3}$ in Fig. 7 and to process $A$ discussed in §6.2:

$$A' = \forall y, w(1; \text{snd}(w, y) \otimes \{w:x\} \rightarrow \text{rcv}(x, y) \| \|P\|_{\pi_3})$$

Assume that in constraint $\text{snd}(y_1, y_2)$ 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:Comβ), we obtain that pattern $\text{snd}(w, y) \otimes \{w:x\}$ has an unrestricted variable ($y$) and two restricted variables, $w$ and $x$. We then infer that $A'$ is not typable because a typing derivation would need to perform an insecure abstraction on the restricted variable $w$. The tree for the failed derivation is as follows, the reasoning can be read in a top-down fashion:

1. $\vdash_\circ \text{rcv}(x, y) \| \|P\|_{\pi_3}$ (L:Abs)
2. $\vdash_\circ \{w\} : \{y\} \vdash \bullet \text{snd}(w, y) \{w, x\}; \{y\} \vdash \bullet \text{snd}(w, y) \otimes \{w:x\}$ (L:Comβ)
3. $\vdash_\circ \forall y, w(1; \text{snd}(w, y) \otimes \{w:x\} \rightarrow \text{rcv}(x, y) \| \|P\|_{\pi_3})$

Notice that the derivation fails since predicate $\{w, y\} \subseteq \{y\}$ will never be true.

## 7 Encoding $s\pi$ with Session Establishment

In this section we consider $s\pi^+$, an extension of $s\pi$ with constructs for session establishment (or session initiation). First, we formalize a translation of $s\pi^+$ into $s\pi$, following the informal translation suggested by Vasconcelos [Vas12]. Then, we translate $s\pi^+$ into $lcc^+$ building upon the translation given in §4 to accommodate a secure phase of session establishment. We show that our extended translation retains the correctness properties of the encoding in §4 (Cor. 7.13), and is well-typed according to the discipline given in §6 (Thm. 7.14). As such, our extended translation enjoys a robust treatment of restriction and scope extrusion, ensured by combining generalized abstractions and secure patterns.

Next we introduce $s\pi^+$ (§7.1), present its translation into $lcc^+$ (§7.2), and establish that this translation is an encoding (cf. Def. 4.2) and is well-typed (§7.3).
7.1 The Calculus $s\pi^+$

The syntax of $s\pi^+$ extends that of $s\pi$ with service requests and accepts, two constructs useful to represent session establishment (or initiation). Our constructs extend those defined in [HKV98] with information on the locations (computation sites) where services reside, as in the distributed $\pi$-calculus [Hen97]. 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 make explicit the fact that services are distributed and that predefined authorization policies govern their interactions.

Formally, we reuse the set $V_{s\pi}$ as the set of variables for $s\pi^+$, which ranges over $x_1, x_2, \ldots$. Also, let $S_{s\pi}$ be a set of service names, ranged over by $a, a', b, b', \ldots$, which will be used in the session initiation phase. We will range over $u, u', v, v'$ to denote elements in $V_{s\pi} \cup S_{s\pi}$. Also, let $m, n, \ldots$ range over locations and $\rho, \rho', \ldots$ denote sets of locations. The syntax of $s\pi^+$ extends that of $s\pi$ (cf. Def. 3.1) with two new constructs:

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

Moreover, we will assume that values $v, v', \ldots$ in $s\pi^+$ denote variables and constants (as in $s\pi$), but also service names in $S_{s\pi}$. This way, an output-prefixied process $\pi.v.P$ can communicate also service names.

The operational semantics for $s\pi^+$ extends the reduction relation given in Fig. 1 with the following rule:

$$\text{[Est]} \ [\pi^m(z).P]^n \mid [a^c_\rho(x).Q]^m \rightarrow_{s\pi} (\nu xy)(P[y/z] \mid Q) \quad (n \in \rho)$$

(4)

With a slight abuse of notation, we will write $\rightarrow_{s\pi}$ to denote a reduction in $s\pi^+$ and $P \equiv_{s\pi} Q$ when two $s\pi^+$ processes are structurally congruent.

Having constructs for service acceptance and request is convenient: they allow us to describe service names and locations, two elements not present in $s\pi$. This way, $s\pi^+$ can be seen as being at a higher abstraction level than $s\pi$. This convenience is useful for modeling, but does not represent an expressiveness gain: we can represent service acceptance and request in $s\pi$. We now define a translation composed of a mapping $\llbracket \cdot \rrbracket^* : s\pi^+ \rightarrow s\pi$ and a renaming policy $\varphi[\cdot]^*$. Mapping $\llbracket \cdot \rrbracket^*$ is in two 'levels'; its formal definition, given below, relies on the set of free service names of a process $P$, denoted $\text{fsn}(P)$. This set is defined as follows:

$$\text{fsn}(0) = \emptyset \quad \text{fsn}(P \mid Q) = \text{fsn}(P) \cup \text{fsn}(Q)$$
$$\text{fsn}(\pi.v.P) = \{ v \} \cup \text{fsn}(P) \quad \text{fsn}(\nu xy)P = \text{fsn}(P)$$
$$\text{fsn}(a^c_\rho(x).Q) = \{ a \} \cup \text{fsn}(Q) \quad \text{fsn}(\ast x(z).P) = \text{fsn}(P)$$
$$\text{fsn}(\pi^m(z).P) = \begin{cases} \{ v \} \cup \text{fsn}(P) & \text{if } v \in S_{s\pi} \\ \text{fsn}(P) & \text{otherwise} \end{cases} \quad \text{fsn}(x \triangleright l.P) = \bigcup_{i \in l} \text{fsn}(P_i)$$
$$\text{fsn}(x(z).P) = \text{fsn}(P) \quad \text{fsn}(x \triangleright l.P) = \text{fsn}(P)$$

Our translation will consider $s\pi^+$ programs, defined as follows:
Notation 7.1 (sπ⁺ Programs). We say that an sπ⁺ process \( P \) is a program if \( \text{fn}(P) \setminus \text{fsn}(P) = \emptyset \). That is, every variable that is not a service name is bound in \( P \).

We may now define:

Definition 7.2 (Translation of sπ⁺ into sπ). We define the translation from sπ⁺ programs into sπ processes as the pair \( (\llbracket \cdot \rrbracket^*, \varphi_{\\lvert \cdot \rangle^+}) \), where:

(a) \( \llbracket \cdot \rrbracket^* \) is the process mapping defined as follows:
\[
\llbracket P \rrbracket^* = (\nu a_1^1, a_2^1) \cdots (\nu a_1^n, a_2^n)(\llbracket P \rrbracket^+)
\]
where \( \text{fsn}(P) = \{a^1, \ldots, a^n\} \) and the auxiliary mapping \( \llbracket \cdot \rrbracket^+: s\pi^+ \rightarrow s\pi \) is defined as
\[
\llbracket [a^n(z)].P \rrbracket^+ = a_1 < m.a_1 < n.a_1(z).\llbracket P \rrbracket^+
\]
\[
\llbracket [a^n(x).Q] \rrbracket^+ = a_2 \triangleright \{ m : a_2 \triangleright \{ l_i : (\nu x_y)(\overline{a_2} y | \llbracket Q \rrbracket^+) \}_{i \in \rho} \}
\]
and as a homomorphism for the other sπ⁺ constructs.

(b) \( \varphi_{\\lvert \cdot \rangle^+} : \mathcal{V}_\pi \cup \mathcal{S}_\pi \rightarrow \mathcal{V}_\pi \) is the function defined as:
\[
\varphi_{\\lvert \cdot \rangle^+}(u) = \begin{cases} (u_1, u_2) & \text{if } u \in \mathcal{S}_\pi \\ u & \text{otherwise} \end{cases}
\]

The two levels are thus defined by two different mappings, denoted \( \llbracket \cdot \rrbracket^* \) and \( \llbracket \cdot \rrbracket^+ \). Given an sπ⁺ process \( P \), mapping \( \llbracket \cdot \rrbracket^* \) extracts the service names in \( P \) and builds an appropriate context of restricted names: each \( a^i \in \text{fsn}(P) \) is “split” into (restricted) names \( a_1^i \) and \( a_2^i \); this context is essential to enable reductions in sπ. Then, mapping \( \llbracket \cdot \rrbracket^+ \) actually translates the source process into an sπ process, exploiting nested selections and branchings.

Due to its compositionality properties, the translation \( \llbracket \cdot \rrbracket^* \) can be shown to be an encoding in a sense that is strictly weaker than that in Def. 4.2—see App. D for details. For simplicity, here we only present the operational correspondence statement associated to this weaker notion:

Theorem 7.3 (Operational Correspondence for \( \llbracket \cdot \rrbracket^* \)). Let \( P \) be an sπ⁺ process. Then:

1. Soundness: If \( P \rightarrow^\pi Q \) then \( \llbracket P \rrbracket^* \rightarrow^k \llbracket P \rrbracket^* \) where \( k = 1 \) or \( k = 3 \).

2. Completeness: If \( \llbracket P \rrbracket^* \rightarrow^\pi R \) then either:
   (a) \( R \equiv^\pi \llbracket Q \rrbracket^* \) (or)
   (b) \( R \rightarrow^2 \llbracket Q \rrbracket^* \).

Proof. The proof proceeds in each numeral by induction on the length of the reduction relation \( \rightarrow^\pi \), followed by a case analysis on the last applied rule (see App. D for details). \( \square \)

The encoding \( \llbracket \cdot \rrbracket^* \) also allows us to reuse the session type system given in §3.1 for sπ⁺ processes. In particular, we define a class of well-formed sπ⁺ programs:

Definition 7.4 (Well-formed sπ⁺ programs). An sπ⁺ program \( P \) is well-formed if \( \vdash_{s\pi} \llbracket P \rrbracket^* \).

This way, well-formed sπ⁺ programs satisfy the properties ensured by the type system in Fig. 3. This follows as a corollary of Lem. 3.8.
7.2 Translating \( s \pi^+ \) into \( 1cc^+ \)

We now present our translation of \( s \pi^+ \) into \( 1cc^+ \). Key novelties with respect to the encoding given in §6 are: first, we consider the session establishment phase with locations; second, to ensure that this phase is done correctly, we translate session declarations/requests by implementing a simple authentication protocol, the well-known Needham-Schroeder-Lowe (NSL) protocol \[Law96\].

7.2.1 A Constraint System for Secure Sessions

In the presence of abstractions with local information (§6.1), processes may query the store about local and global constraints. It is crucial to avoid publishing local (restricted) information (e.g., session identifiers, encryption keys, nonces) in the global store. To this end, our translation of \( s \pi^+ \) into \( 1cc^+ \) relies on a security constraint system that combines local and global information with basic cryptographic primitives. This is another instantiation of Def. 3.17 which builds upon similar constraint systems given in [OV08a, OV08b, HL09].

Definition 7.5 (Security Constraint System). Let \( \Sigma \) and \( \Delta \) be the function symbols and predicates given in Fig. 13. The security constraint system is the tuple \( \langle C, \Sigma \cup \Delta, \vdash_C \rangle \), where \( C \) is the set of all constraints obtained by using linear operators \( !, \otimes \) and \( \exists \) over the functions of \( \Sigma \) and predicates of \( \Delta \), and where \( \vdash_C \) is given by the rules in Fig. 4 extended with the non-logical axioms in Fig. 14.

We comment on the signatures for functions and predicates given in Fig. 13 which differ from those in Fig. 6 in several respects. First, \( \Sigma \) includes functions for handling encryption (used to model session initiation) and tuples: given a location \( n \) as an (unrestricted) argument, functions \( p, r \), and \( s \) return the public, restricted (i.e., private), and symmetric key of \( n \), respectively. We use \( k \) to range over function identifiers \( p, r \), and \( s \) and write \( k^{-1} \) to denote inverse of \( k \), defined as \( s^{-1} = s \) and \( p^{-1} = r \), as expected. Function \( \text{enc}(k; m) \) returns message \( m \) encrypted with a key \( k \), which is restricted. Decryption is enabled via entailment/deduction in the constraint system (Fig. 14). Lastly, function \( \text{tup}_j(x) \) models a tuple of \( j \) (unrestricted) elements \( (j \geq 1) \).

As before, predicates in \( \Delta \) are used to represent session communication; in this case, we exploit the distinction between restricted and unrestricted variables. Predicate \( \text{snd}(x; y) \) takes two arguments: a (restricted) session key and an unrestricted message. That is, while the communication subject should be private to the session, its associated communication object can be publicly available. Predicate \( \text{rcv}(x; y; \epsilon) \) models the acknowledgement of \( \text{snd} \), and contains the session key and the value. In this case, both subject and object are restricted. Predicates \( \text{sel} \) and \( \text{bra} \) model communication of a label object, and so they are conceptually similar to \( \text{snd} \) and \( \text{rcv} \). Predicate \( \{x; y\} \) declares \( x, y \) as co-variables; both endpoints are restricted. Novelties with respect to the predicates in Fig. 6 are \( \text{loc}_\rho(\epsilon; x) \), \( \text{ch}(x; \epsilon) \), and \( \text{out}(\epsilon; m) \):

- Predicate \( \text{loc}_\rho(\epsilon; x) \) asserts that location \( x \) is in the set of locations \( \rho \).
- Predicate \( \text{ch}(x; \epsilon) \) declares \( x \) as a channel.
- Predicate \( \text{out}(\epsilon; m) \) indicates that a message \( m \) has been sent through a public, potentially insecure medium; it is solely used to model service agreement during session initiation.

The following notation will be useful in processes.

Notation 7.6. Function \( \text{enc}(k; x) \) will be written as \( \{x\}_k \). Also, tuple \( \text{tup}_n(\epsilon; x_1, \ldots, x_n) \), with \( n \geq 1 \), will be written \( \langle x_1, \ldots, x_n \rangle \).

\[^1\] This choice is orthogonal to the translation; other, more sophisticated protocols could be considered.
$$\Sigma \overset{\text{def}}{=} p(e; n) \mid r(e; n) \mid s(e; n) \mid \text{enc}(k; m) \mid \text{tup}_j(e; x)$$

$$\Delta \overset{\text{def}}{=} \text{snd}(x; y) \mid \text{rev}(x; y; e) \mid \text{sel}(x; l) \mid \text{bra}(x; l; e) \mid \{x;y\} \mid \text{loc}_p(e; x) \mid \text{ch}(x; e) \mid \text{out}(e; x)$$

Figure 13: Security constraint system: Function and predicate symbols.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E:Loc)</td>
<td>$c \vdash_C \text{out}(e; n); n \in \rho$</td>
<td>$c \vdash_C \text{loc}_p(e; n)$</td>
</tr>
<tr>
<td>(E:Cov-Comm)</td>
<td>$c \vdash_C {x;y}$</td>
<td>$c \vdash_C \text{ch}(x) \quad c \vdash_C {x;y}; x \neq y$</td>
</tr>
<tr>
<td>(E:Cov)</td>
<td>$c \vdash_C \text{ch}(x)$</td>
<td>$c \vdash_C \text{ch}(y)$</td>
</tr>
<tr>
<td>(E:Enc)</td>
<td>$c \vdash_C \text{out}(e; k)$</td>
<td>$c \vdash_C \text{out}(e; m) \otimes \text{out}(e; \text{enc}(k(x); m))$</td>
</tr>
<tr>
<td>(E:Dec)</td>
<td>$k \in {s, p} \quad c \vdash_C \text{out}(e; k^{-1}(x)) \quad c \vdash_C \text{out}(e; \text{enc}(k(x); m))$</td>
<td></td>
</tr>
<tr>
<td>(E:Tup)</td>
<td>$\forall i \in {1, \ldots, j}; c \vdash_C \text{out}(e; e_i)$</td>
<td>$c \vdash_C \text{out}(e; \text{tup}_j(e; e_1, \ldots, e_j))$</td>
</tr>
<tr>
<td>(E:Proj1)</td>
<td>$c \vdash_C \text{out}(e; \text{tup}_j(e; e_1, \ldots, e_i, \ldots, e_j))$</td>
<td></td>
</tr>
<tr>
<td>(E:Proj2)</td>
<td>$c \vdash_C \text{out}(e; \text{tup}_j(e; e_1, \ldots, e_i, \ldots, e_j))$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Security constraint system: Non-logical axioms.

We now comment on the entailment rules given in Fig. 14. Rule (E:Loc) is used to verify that a location belongs to a set $\rho$; if the location is known (out($e; n$)) and it belongs to a set $\rho$ then we can obtain predicate loc$_p(e; n)$. Rule (E:Cov) relates the two endpoints, known only to the participants of that session: it states that given an endpoint id $\text{ch}(x; e)$ and the co-variable constraint, we may obtain the id for the other endpoint $y$. (E:Cov-Comm) states that the co-variable constraint is commutative. Rule (E:Enc) gives the key of a message. 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. Rules (E:Proj1) and (E:Proj2) handle tuples: while Rule (E:Proj1) allows us to project individual elements, discarding the remaining elements, Rule (E:Proj2) allows to project any element of the tuple while preserving the remaining tuple elements.

### 7.2.2 The Translation

We now introduce the translation of $s\pi^+$ into $\text{1cc}^+$. As explained in §6, 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 next; it is parameterized by a set of pairs of co-variables, denoted $f$.

**Definition 7.7 (Translation of $s\pi^+$ into $\text{1cc}^+$).** We define the translation from $s\pi^+$ into $\text{1cc}^+$ as the pair $(\llbracket \cdot \rrbracket_f^\Pi, \cdot_{\Pi \Pi}^\Pi)$, where:

(a) $\llbracket \cdot \rrbracket_f^\Pi$ is the process mapping defined in Fig. 15.
The translation in Fig. 15 is similar to the encoding of $\pi$ into $\text{cc}$ (cf. Fig. 7). Two main differences concern the authentication protocol and local information:

- Session establishment is implemented following the NSL protocol. A process request starts by sending a tuple containing a nonce $w$ and the location where the requester resides ($n$). The tuple is encrypted using the public key of the location where the requested service resides ($m$). The requested service then creates two fresh endpoints ($x, y$) and receives an encrypted tuple, containing $w$ and $n$. Notice that it is necessary to decrypt this tuple to extract location $n$. This is represented by the guard being constraint $\text{out}(c, \{\langle z, n \rangle \}_{p(m)}) \otimes \text{loc}_{p}(c, n)$, which verifies that the location is indeed allowed to access the service, and that the encrypted tuple has the correct structure and content. After that, the requested service encrypts (using the public key of $n$) and sends a tuple containing the nonce $w$, endpoints $x, y$ and its own location ($m$). Lastly, the requester receives, decodes, sends back endpoints $x, y$, encoded using the public key of $m$ to acknowledge that it has received them, thus declaring that $x, y$ will indeed be co-variables.

- The translation in Fig. 15 uses abstractions with local information and secure patterns. Within session communications, we require the knowledge of being a channel to be private ($\text{ch}(x; c)$). This is used in conjunction with the public constraint $\{x: f_x\}$ to avoid interferences. Generated after session establishment, covariable constraints are collected in the set $f$. This is made explicit in the translations of $(\nu xy)P$ and the session establishment constructs. In Fig. 15, 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 $f_y = x$.

### 7.3 Correctness of the Translation

We now state correctness of the translation $\mathcal{E}^{\pi}_{\text{cc}} : \pi^+ \to \text{cc}^+$, in the sense of Def. 4.2. We mostly build upon the approach given in §4.3.1 and §4.3.2. The notion of evaluation context is as in Def. 4.7. We also establish typability of encoded processes (Thm. 7.14).

**Theorem 7.8 (Name invariance for $\mathcal{E}^{\pi}_{\text{cc}}$).** Let $P$ be a well-formed $\text{ss}^+$ program (cf. Def. 7.4). Also, let $\sigma$ and $x$ be a substitution satisfying the renaming policy for $\mathcal{E}^{\pi}_{\text{cc}}$ (Def. 7.7(b)) and a variable in $\text{cc}$, resp. Then $\mathcal{E}^{\pi}_{\text{cc}}[\sigma P] = \mathcal{E}^{\pi}_{\text{cc}}[P]^{\pi}$, where $\varphi_{\pi}^{\text{cc}}(\sigma(x)) = \sigma'(\varphi_{\pi}^{\text{cc}}(x))$ and $\sigma = \sigma'$.

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

**Theorem 7.9 (Compositionality of $\mathcal{E}^{\pi}_{\text{cc}}$).** Let $P$ be a well-formed $\text{ss}^+$ program (cf. Def. 7.4). Also, let $E[\cdot]$ be an $\text{ss}^+$ evaluation context (cf. Def. 4.7). Then we have: $\mathcal{E}^{\pi}_{\text{cc}}[E[P]] = \mathcal{E}^{\pi}_{\text{cc}}[\mathcal{E}^{\pi}_{\text{cc}}[P]]$.

Before stating our operational correspondence result, we adapt the definitions of continuation and intermediate processes (Def. 4.10 and Def. 4.11) to capture the constructs in $\pi^+$:

**Definition 7.10 (Continuation processes for $\pi^+$).** Let $P$ be an $\pi^+$ process such that $P \equiv_\pi (\nu x \bar{y})Q(x, Q | R)$ or $P \equiv_\pi (\nu x \bar{y})Q(x, l.Q | R)$ or $P \equiv_\pi [\pi^m (x_i).Q]^n | R$, for some $Q, R, \bar{y}, l, a$. Assume $x_i \in \bar{x}, y_i \in \bar{y}$ are
The set of intermediate processes of $\llbracket P \rrbracket_f$, denoted $\llbracket P \rrbracket_f$, is defined as

$$\llbracket P \rrbracket_f = \{ S | \llbracket P \rrbracket_f \xrightarrow{\tau} S = \llbracket C \rrbracket_f \llbracket R_1 \rrbracket_f,\ldots,\llbracket R_i \rrbracket_f,\ldots,\llbracket R_n \rrbracket_f \}$$

with 1 ≤ i ≤ n.

We write $\llbracket P \rrbracket$ when the co-variable $y_i$ is unimportant.

Notice that the extension of Lem. 4.9 to $s\pi^+$ and $\llbracket \cdot \rrbracket_f^S$ holds as well, since the new constructs for $s\pi^+$ do not introduce new evaluation contexts. Considering this, we may now define:

**Definition 7.11 (Intermediate Processes for $s\pi^+$).** Let $P$ be a well-formed $s\pi^+$ program (cf. Def. 7.4), with translated form (cf. Lem 4.9)

$$\llbracket P \rrbracket_f^S = \llbracket C \rrbracket_f^S \llbracket R_1 \rrbracket_f^S,\ldots,\llbracket R_i \rrbracket_f^S,\ldots,\llbracket R_n \rrbracket_f^S$$

(with 1 ≤ i ≤ n).

The set of intermediate processes of $\llbracket P \rrbracket_f^S$, denoted $\llbracket P \rrbracket_f^S$, is defined as

$$\llbracket P \rrbracket_f^S = \{ S | \llbracket P \rrbracket_f^S \xrightarrow{\tau} S = \llbracket C \rrbracket_f^S \llbracket R_1 \rrbracket_f^S,\ldots,\llbracket R_i \rrbracket_f^S,\ldots,\llbracket R_n \rrbracket_f^S \}$$


Finally, we state operational correspondence:

**Theorem 7.12 (Operational Correspondence for $\llbracket \cdot \rrbracket^\mathbf{s}$).** Let $P, Q$ be well-formed $s\pi^+$ programs (cf. Def. 7.4) and $R, S$ be $\mathbf{lcc}^+$ processes. Then:

1. **Soundness:** If $P \rightarrow_{\pi} Q$ then one of the following holds:
   a. $\llbracket P \rrbracket^\mathbf{s} \rightarrow_1 R$, for some $R$ such that $R \approx \llbracket Q \rrbracket^\mathbf{s}$.
   b. (or) $\llbracket P \rrbracket^\mathbf{s} \equiv_1 S' \rightarrow_2 R'$, for some $R', S'$, $R$ such that $R \approx \llbracket Q \rrbracket^\mathbf{s}$.
   c. (or) $\llbracket P \rrbracket^\mathbf{s} \rightarrow_3 R$, for some $R$ such that $R = \llbracket Q \rrbracket^\mathbf{s}$.

2. **Completeness:** If $\llbracket P \rrbracket^\mathbf{s} \rightarrow_2 S$ then one of the following holds:
   a. $P \rightarrow_{\pi} Q$, for some $Q$ and $\llbracket Q \rrbracket^\mathbf{s} \approx S$.
   b. (or) $S \in \llbracket P \rrbracket^\mathbf{s}$ and, for some $S'$ and $Q$, we have that $S \rightarrow_1 S'$, $P \rightarrow_{\pi} Q$, and $\llbracket Q \rrbracket^\mathbf{s} \approx S'$.
   c. (or) $S \in \llbracket P \rrbracket^\mathbf{s}$ and $S \rightarrow_2 S'$, for some $S'$ and $Q$, we have that $P \rightarrow_{\pi} Q$ and $\llbracket Q \rrbracket^\mathbf{s} = S'$.

We discuss some differences with respect to the case of $s\pi$. The above theorem adds a new possibility for both soundness and completeness (cases (c)), which takes into account reduction(s) due to session establishment. Since the NSL protocol is a 3-step protocol, three reductions in $\mathbf{lcc}^+$ are needed to mimic it. In general, adding a session establishment phase does not affect the operational correspondence results between $s\pi^+$ and $\mathbf{lcc}^+$.

Based on the above theorems, we may state:

**Corollary 7.13.** Translation $\langle \llbracket \cdot \rrbracket^\mathbf{s}, \varphi \rangle_{\llbracket \cdot \rrbracket^\mathbf{s}}$ is an encoding (Def. 4.3).

Our final correctness property for the translation is typability with respect to the type system in §6. We first introduce an useful notation: In type derivations we will use the following abbreviations for predicates and functions:

\[
\begin{align*}
\text{out} &= o \\
\text{loc}_p(\epsilon; n) &= l^p_n \\
r(l) &= r_l \\
p(l) &= p_l
\end{align*}
\]

**Theorem 7.14 (Typability of $\llbracket \cdot \rrbracket^\mathbf{s}$).** Let $P$ be a well-formed $s\pi^+$ program (cf. Def. 7.4). Then $\vdash_o \llbracket P \rrbracket^\mathbf{s}$.

**Proof (Sketch).** By induction on the structure of $P$ (see App. D.4 for details). As an example, Fig. 16 gives the derivation tree for the case $P = \{a_0(x).Q\}^m$.

The above theorem attests that, provided a disciplined used of patterns (following the signature in Fig. 13), our encoding adheres to a robust interpretation of restriction and scope extrusion. By using secure patterns in our encoding $\llbracket \cdot \rrbracket^\mathbf{s}$, 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 7.14 with Theorems 6.2 and 7.12 (type preservation and operational correspondence, respectively) formalizes static and dynamic robustness guarantees for our declarative representations of structured communications.
lcc naturally matches the linear communication in between operational and declarative settings, as we have done here. We have shown that the linearity of

thermore, the developments in [LOP10] do not explore the transfer of reasoning and analysis techniques using unrestricted types in non-deterministic behavior (as required for session establishment). In contrast, exploiting linearity, our

gave an interpretation of intuitionistic linear logic as session types, in the style of Curry-Howard [CP10].

in [Hae11, MM12] are not concerned with the analysis of communication-centric systems in general, nor with session-based concurrency in particular.

The relationship between linear logic and session types has been recently clarified. Caires and Pfennig gave an interpretation of intuitionistic linear logic as session types, in the style of Curry-Howard [CP10].

\begin{figure}[h]
\centering
\begin{itemize}
\item \begin{prooftree}
\hypo \{Q\}_{P}
\hypo \{y\}_{m}
\hypo o((x,y)_{m})
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\item \begin{prooftree}
\hypo \forall \text{c}(\exists \; x, y, m)
\hypo \forall \text{c}(\exists \; (z, n))
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\item \begin{prooftree}
\hypo \exists \; x, y, m
\hypo \forall \text{c}(\exists \; z, n)
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\item \begin{prooftree}
\hypo \forall \text{c}(\exists \; z, n)
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\item \begin{prooftree}
\hypo \forall \text{c}(\exists \; (z, n))
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\item \begin{prooftree}
\hypo \forall \text{c}(\exists \; (z, n))
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\item \begin{prooftree}
\hypo \forall \text{c}(\exists \; (z, n))
\hypo \vdash \ imply \text{(L-Pre)}
\end{prooftree}

\end{itemize}

8 Related Work

Our developments build upon the spirit of previous works by the current authors [LOP10, HL09]. However, because of the substantial technical differences (notably, the presence of linearity) our results cannot be derived from those in [LOP10] (which developed encodings of session $\pi$-calculi in utcc) nor in [HL09] (which presented a type system for utcc).

A key difference with respect to [LOP10] is the ccp language considered (1cc here, utcc in [LOP10]): this is crucial because, as already discussed, thanks to the linear abstractions in 1cc, our encodings of $\pi\pi$ and $\pi\pi^+$, presented in §4 and §7, are rather compact and satisfy tight operational correspondences. We also improve on expressiveness: since utcc is a deterministic calculus, the encoding in [LOP10] cannot capture non-deterministic behavior (as required for session establishment). In contrast, exploiting linearity, our encoding captures non-deterministic session establishment and also forms of non-determinism derivable using unrestricted types in $\pi\pi$. Fig. 8 gives a process encodable in our approach but not in [LOP10]. Furthermore, the developments in [LOP10] do not explore the transfer of reasoning and analysis techniques between operational and declarative settings, as we have done here. We have shown that the linearity of 1cc naturally matches the linear communication in $\pi\pi$. In utcc abstractions are persistent, and so the encoding in [LOP10] is more involved and its operational correspondence is harder 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 [LOP10]. The type system in [HL09] (defined for utcc) and the one in §6 are similar in spirit, but not in details: moving to 1cc and considering linearity requires non-trivial modifications.

Haemmerlé [Hae11] gives an encoding of an asynchronous $\pi$-calculus into 1cc, and establishes its operational correspondence. Since his encoding concerns two asynchronous models, this operational correspondence is rather direct. Monjaz and Marín [MM12] encode the asynchronous $\pi$-calculus into Flat Guarded Horn Clauses. They consider compositionality and operational correspondence issues, as we do here. In contrast to [Hae11, MM12], here we consider a session $\pi$-calculus with synchronous communication, which adds challenges in the encoding and its associated correctness proofs. The developments in [Hae11, MM12] are not concerned with the analysis of communication-centric systems in general, nor with session-based concurrency in particular.

The relationship between linear logic and session types has been recently clarified. Caires and Pfennig gave an interpretation of intuitionistic linear logic as session types, in the style of Curry-Howard [CP10].
Wadler developed this interpretation for classical linear logic [Wad12]. Giunti and Vasconcelos gave a linear reconstruction of session types [GV10]; their system is further developed in [Vas12].

Loosely related to our work are [BHTY10, CGHL10]. Bocchi et al. [BHTY10] integrate declarative requirements into multiparty session types by enriching (type-based) communication descriptions with logical assertions which are globally specified within multiparty protocols and potentially projected onto specifications for local participants. Rather than a declarative process model based on constraints, the target process language in [BHTY10] is a \( \pi \)-calculus augmented with predicates for checking (both outgoing and incoming) communications. It should be interesting to see if such an augmented session \( \pi \)-calculus can be encoded in \( lcc \) by extending the encoding we have presented in \( \S 4 \). Also in the context of choreographies, although in a different vein, Carbone et al. [CGHL10] explore reasoning via a variant of Hennessy-Milner logic for global specifications.

Several works have aimed at combining declarative and operational descriptions of services. Works on Web service contracts have been particularly successful at combining operational descriptions (akin to CCS specifications) and constraints, where the entailment of a constraint represents the possibility for a service to comply with the requirements of a requester. In [BM07b, BM11], Buscemi and Montanari develop CC-pi, a constraint language that combines the message-passing communication model from the \( \pi \)-calculus with operations over a constraint store as in \( ccp \) languages. Analysis techniques for CC-pi processes exploit behavioral equivalences (open bisimulation [BM08a]); logical characterizations of process behavior have not been studied. A challenge for obtaining such characterizations is CC-pi’s \textit{retract} construct, which breaks the monotonicity requirements imposed for constraint stores in the \( ccp \) model. We do not know of any attempts on applying session-type analysis for specifications in CC-pi.

In a similar line of work, Coppo and Dezani-Ciancaglini [CD09] present an extension of the session \( \pi \)-calculus in [HVK98] with constraint handling operators, such as tells, asks and constraint checks. Session initiation is then bound to the satisfaction of constraint in the store. The merge of constraints and a session type system guarantees \textit{bilinearity}, i.e. channels in use remain private, and that the communications proceed according to the order prescribed by the session type. It is worth noticing that the underlying constraint store in [CD09] is not linear, which can create potential races among different service providers. A linear treatment of constraints (or a process construct similar to CC-pi’s retract) is left for future work.

The interplay of constraints and service contracts has been also studied by Buscemi et al. [BCDM11]. In their model, service interactions follow three phases: service negotiation, commitment and service execution. In a service negotiation phase, processes agree on fulfilling certain desired behaviours, without guarantee of success. Once committed, it is guaranteed that the execution of processes will honour promised behaviours, and forbidding a service to get stuck (deadlock-freedom). The model in [BCDM11] uses two languages: a variant of CCS is used as a source language, where the behaviour of services and clients is specified; these specifications are later compiled to a target language based on CC-pi with no retraction operator, where constraints guarantee that interactions between clients and services do not deadlock. We believe that this two-level model could be enriched by the use of linear constraints similar to the ones studied in \( lcc \) and presented in this paper, thus refining the consumption of resources in the environment.

The work of Bartoletti et al. [BZ10, BTZ12] promotes contract-oriented computing as a novel vision for the runtime enforcement of service behaviours. The premise is that in scenarios where third-party components can be used but not inspected, verification based on (session) types becomes a challenge. Contracts exhibit promises about the expected runtime behaviour of each component; they can be used to establish new sessions (contract negotiation) and to enforce that components abide to their promised behaviour (honesty). The calculus for contracting processes is based on PCL, a propositional contract logic that includes a contractual form of implication [BZ10]; this enables to express multiparty assume-guarantee specifications where services only engage in a communication once there are enough guarantees.
that their requirements will be fulfilled. PCL is used as the underlying constraint system for the contract language used in [BZ10], a variant of cc with name-passing primitives. In its accompanying Technical Report [BZ09], the authors analyse the expressive power of the contract calculus with respect to the synchronous π-calculus, establishing name-invariance, compositionality and operational correspondence, as in the spirit of our work. In later developments [BTZ12] the authors introduce CO₂, a generic framework for contract-oriented computing. A characterization of contracts as processes and as formulae in PCL has been developed. It will be interesting to explore linear variants of PCL, to map the linearity conditions of session-based communication.

9 Concluding Remarks

In this work, we have presented different interpretations (encodings) of session π-calculi into lcc, a declarative process model based on the ccp paradigm. Our encodings crucially exploit linearity, a common trait in both models. Linearity enables us to define intuitive translations of session-based processes and to obtain precise correctness properties for them (notably, operational correspondence), improving our previous work [LOP10].

Our study covers several relevant concerns in the modelling and analysis of communication-centric systems—see Fig. 17. Our first encoding, given in §4.2 concerns sπ, the session π-calculus in [Vas12]. The second encoding, given in §5.2.2 concerns πCH, the typed π-calculus derived from the Curry-Howard interpretation of session types as linear logic propositions [CP10, CPT15]. Finally, the third encoding, given in §7.2 considers as source language sπ⁺, the extension of sπ with constructs for session establishment; the target language is 1cc⁺, the extension of lcc with abstractions with local information. This last encoding allows us to exploit another contribution of our work, namely a type system for lcc⁺ processes that enforces secure abstractions, thus addressing an anomaly of known abstraction-based representations of scope extrusion in the π-calculus. In all three cases, we address the correctness of syntactic translations via an abstract notion of encoding, following [Gor10].

Moreover, to further highlight the significance of our encodings, we have exploited them to transfer analysis techniques between operational and declarative paradigms. The transfer is bidirectional: we have shown how an observational equivalence for lcc can be used to define a behavioral equivalence for well-typed sπ processes, and how a type system for deadlock-free session processes can be exploited to identify new classes of suspension-free lcc processes. In both cases, our encodings and their correctness properties were instrumental to formalize the transfer of techniques. We do not know of related works in which a similar transfer of techniques is effectively accomplished.

In future work, we wish to deepen on the integration of operational and declarative approaches. In
particular, we plan to extend our encodings to consider the session $\pi$-calculus with asynchronous (queue-based), eventful semantics defined in [KYH11]. Also, since in this work we have focused on $\pi$-calculus implementations for binary session types, we plan to address declarative interpretation of multiparty session processes [HYC16, BHTY10], so as to have rich declarative interpretations of communication scenarios with more than two participants.

Acknowledgments. We thank Rémy Haemmerlé, Carlos Olarte, Frank Valencia, Vasco Vasconcelos, and Roberto Zunino for useful exchanges, and to the PPDP’15 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, Statically (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 (NOVA LINCS), Universidade Nova de Lisboa, Portugal (Ref. UID/CEC/04516/2013).

References


A Appendix to Section 4

A.1 Static Correctness of $J \cdot K_s π$

We present the proofs for the static properties of the encoding.

Theorem 4.6 (Name Invariance of $J \cdot K_s π$). Statement on Page 21.

Proof. The proof for this theorem proceeds by induction on the structure of $P$ as follows. We only show the cases for $P = 0$ and $P = P = \bar{x}.v.Q$, all the other cases are solve in similar ways:

Case $P = 0$:

(i) By Fig. 7 we have that $[0]_{sπ} = T$.

(ii) Since there are no variables in $T$ then $Tσ = T$.

(iii) By (i),(ii) we have that $[0σ]_{sπ} = [0]_{sπ}σ$.

Case $P = \bar{x}.v.Q$:

(i) By Fig. 7 and Def. 4.4(b) we have that:

$[\bar{x}.v.Qσ]_{sπ} = \text{snd}(x,v)σ \parallel ∀z((rcv(z,v) \otimes \{x : z\})σ → [Qσ]_{sπ})$

(ii) By Fig. 7 and Def. 4.4(b) we have that:

$[\bar{x}.v.Qσ]_{sπ} = \text{snd}(x,v)σ \parallel ∀z((rcv(z,v) \otimes \{x : z\})σ → [Qσ]_{sπ})$

(iii) By the inductive hypothesis (IH) we have that $[Q]_{sπ} = [Qσ]_{sπ}$.

(iv) By (i),(ii),(iii) we have that $[\bar{x}.v.Qσ]_{sπ} = [\bar{x}.v.Qσ]_{sπ}$.

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

Theorem 4.8 (Compositionality of $[\cdot]_{sπ}$). Statement on Page 22.

Proof. The proof proceeds by induction on the structure of $P$ and a case analysis on the grammar in Def. 4.7. We only show the cases for $P = 0$ and $P = P = \bar{x}.v.Q$, all the other cases are solve in similar ways:

Case $P = 0$

• Subcase $E[\cdot] = R | .$

  (i) By Fig. 7 $[E[0]_{sπ} = [R]_{sπ} \parallel [0]_{sπ}$.

  (ii) By (i) and Fig. 7 we have that $[R]_{sπ} \parallel [0]_{sπ} = [E]_{sπ}[0]_{sπ} = [E]_{sπ}$.

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

• Subcase $E[\cdot] = (νx)y(\cdot)$. By Fig. 7 and structural congruence see that $(νx)y(0) \equiv_π 0$, thus $[E]_{sπ}[0]_{sπ} = [E]_{sπ}$.

Case $P = \bar{x}.v.Q$:

• Subcase $E[\cdot] = R | .$

53
A.2 Dynamic Correctness of $\llbracket \cdot \rrbracket_{\text{ss}}$

**Theorem 4.12 (Operational Correspondence).** Statement on Page 24.

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

1. **Soundness:** The proof is by induction on the reduction for $\text{ss}$. We make a case analysis on the last applied rule:

   **Case Rule $[\text{IrT}]$:**
   
   (i) Assume $P = \text{tt}\? (P') : (P'').$
   
   (ii) By (i) then $P \rightarrow_{\pi} P'$ using Rule $[\text{IrT}]$.

   (iii) Then by the application of the definition of $\llbracket \cdot \rrbracket_{\text{ss}}$ (Def. 4.4):
   
   $$\llbracket P \rrbracket_{\text{ss}} = \forall \epsilon (1 \rightarrow \llbracket P' \rrbracket_{\text{ss}}) \parallel \forall \epsilon (1 \rightarrow \llbracket P'' \rrbracket_{\text{ss}})$$

   (iv) By using Rule (C:Sync) (Fig. 5), with $c = 1$ we have the following (note that $\parallel 1 = 1$):
   
   $$\llbracket P \rrbracket_{\text{ss}} \rightarrow_{1} \llbracket P' \rrbracket_{\text{ss}} \parallel \forall \epsilon (1 \rightarrow \llbracket P'' \rrbracket_{\text{ss}})$$

   (v) By (iv) note that the process $\forall \epsilon (1 = 0 \rightarrow \llbracket P'' \rrbracket_{\text{ss}})$ is blocked, the constraint $1 = 0$ cannot be satisfied. Then, conclude that $\llbracket P' \rrbracket_{l} \approx \llbracket P' \rrbracket_{l} \parallel \forall \epsilon (1 = 0 \rightarrow \llbracket P'' \rrbracket_{l})$.

   **Case Rule $[\text{IrF}]$:** Analogous to previous case.

   **Case Rule $[\text{Com}]$:**
   
   (i) Assume $P = (\nu xy)(\pi v.P' \mid y(z).P'')$.
   
   (ii) By (i) $P \rightarrow_{\pi} (\nu xy)(P' \mid P'' \{v/z\})$ using Rule $[\text{Com}]$.

   (iii) By definition of $\llbracket \cdot \rrbracket_{\text{ss}}$:
   
   $$\llbracket P \rrbracket_{\text{ss}} = \exists x, y. (\llbracket \pi v.P' \rrbracket_{\text{ss}} \parallel \forall z ((rcv(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{ss}}) \parallel \forall z, w(snd(w, z) \otimes \{w:y\}) \rightarrow (rcv(y, z) \parallel \llbracket P'' \rrbracket_{\text{ss}}))$$

   All the other cases proceed in the same way as the previous case. □
(iv) By using structural congruence and reduction in \(1cc\) we may build the following reduction:

\[
\begin{align*}
\llbracket P \rrbracket_{\text{sr}} &\equiv_1 \exists x, y. ((\{x:y\} \otimes \text{snd}(x, v)) \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \\
&\quad \forall z, w(\text{snd}(w, z) \otimes \{w:y\}) \rightarrow (\text{rcv}(y, z) \parallel \llbracket P'' \rrbracket_{\text{sr}}) \\
&\quad \rightarrow_1 \exists x, y. ((\{x:y\} \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \text{rcv}(y, v) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}}) \\
&\equiv_1 \exists x, y. ((\{x:y\} \otimes \text{rcv}(y, v)) \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}}) \\
&\rightarrow_1 \exists x, y. ((\{x:y\} \parallel \llbracket P'(y/z) \rrbracket_{\text{sr}}) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}})
\end{align*}
\]

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

\[
\llbracket (\nu xy)(P' \parallel P''(v/z)) \rrbracket_{\text{sr}} = \exists x, y. ((\{x:y\} \parallel \llbracket P'(y/z) \rrbracket_{\text{sr}}) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}})
\]

Case Rule | REPL: Analogous to case Rule | COM], as follows:

(i) Assume \(P = (\nu xy)(\exists v. P' \parallel \star y(z), P'')\).
(ii) By (i) \(P \rightarrow_\pi (\nu xy)(P' \parallel P''(v/z) \parallel \star y(z), P'')\) using Rule | REP|.
(iii) By definition of \(\llbracket \cdot \rrbracket_{\text{sr}}:

\[
\begin{align*}
\llbracket P \rrbracket_{\text{sr}} &\equiv_1 \exists x, y. ((\{x:y\} \parallel \text{snd}(x, v)) \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \\
&\quad !\forall z, w(\text{snd}(w, z) \otimes \{w:y\}) \rightarrow (\text{rcv}(y, z) \parallel \llbracket P'' \rrbracket_{\text{sr}}) \\
&\equiv_1 \exists x, y. ((\{x:y\} \parallel \text{snd}(x, v)) \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \\
&\quad \forall z, w(\text{snd}(w, z) \otimes \{w:y\}) \rightarrow (\text{rcv}(y, z) \parallel \llbracket P'' \rrbracket_{\text{sr}}) \\
&\quad !\forall z, w(\text{snd}(w, z) \otimes \{w:y\}) \rightarrow (\text{rcv}(y, z) \parallel \llbracket P'' \rrbracket_{\text{sr}})
\end{align*}
\]

(iv) Let \(R = !\forall z, w(\text{snd}(w, z) \otimes \{w:y\}) \rightarrow (\text{rcv}(y, z) \parallel \llbracket P'' \rrbracket_{\text{sr}})\). By using the rules of structural congruence and reduction of \(1cc\) one can build the following reduction:

\[
\begin{align*}
\llbracket P \rrbracket_{\text{sr}} &\equiv_1 \exists x, y. ((\{x:y\} \otimes \text{snd}(x, v)) \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \\
&\quad \forall z, w(\text{snd}(w, z) \otimes \{w:y\}) \rightarrow (\text{rcv}(y, z) \parallel \llbracket P'' \rrbracket_{\text{sr}}) \parallel R \\
&\rightarrow_1 \exists x, y. ((\{x:y\} \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \\
&\quad \text{rcv}(y, v) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}}) \parallel R \\
&\equiv_1 \exists x, y. ((\{x:y\} \otimes \text{rcv}(y, v)) \parallel \forall z((\text{rcv}(z, v) \otimes \{x:z\}) \rightarrow \llbracket P' \rrbracket_{\text{sr}}) \parallel \\
&\quad \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}} \parallel R \\
&\rightarrow_1 \exists x, y.((\{x:y\} \parallel \llbracket P'(y/z) \rrbracket_{\text{sr}}) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}} \parallel R)
\end{align*}
\]

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

\[
\llbracket (\nu xy)(P' \parallel P''(v/z)) \parallel \star y(z), P'' \rrbracket_{\text{sr}} = \exists x, y. ((\{x:y\} \parallel \llbracket P'(y/z) \rrbracket_{\text{sr}}) \parallel \llbracket P''(v, x/z, w) \rrbracket_{\text{sr}} \parallel R)
\]

Case Rule | SEL|: Analogous to case Rule | COM|.

55
Completeness: The proof proceeds directly by appealing to the structure of an encoded well-formed, typable program (Lem. 4.9, Not. 3.7). Without loss of generality, we can reduce the proof to the minimal completeness.

(i) By considering the encoded form of a well-formed program (Lem. 4.9) we have:
\[
\llbracket P \rrbracket_{\pi \tau} \equiv \exists \bar{x}, \bar{y}. (\llbracket R_1 \rrbracket_{\pi \tau} \mid \ldots \mid \llbracket R_n \rrbracket_{\pi \tau} \mid V)
\]
where each \( R_i, 0 \leq i \leq n \) is a pre-redex.

(ii) From the hypothesis, if \( P \not\rightarrow_1 1 \) then it is vacuously true.

(iii) From (i), we know that each \( R_i \) can reduce: we consider the cases where there is only one and two pre-redexes and they can reduce. Without loss of generality, consider the cases where there is only one pre-redex.

Case \( R_i = v? (R'_i); (R''_i) \) (there exists only one pre-redex):

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

• Subcase \( v = \text{tt} \):
  (i) We show that \( R_i \rightarrow_{\pi} R'_i \) by using Rule [IrT].
  (ii) Conclude by setting \( Q = R'_i \) (Thus showing such \( Q \) exists). Note that

\[
\llbracket R'_i \rrbracket_{\pi \tau} \mid \forall \epsilon(1 \rightarrow \llbracket P'' \rrbracket_{\pi \tau}) \approx \llbracket R'_i \rrbracket_{\pi \tau}
\]

• Subcase \( v = \text{ff} \): Analogous to the previous case.

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

Subcase Input/output interaction:

(i) Assume \( R_i = \pi.v.R'_i \) and \( R_j = y(z).R'_j \),

(ii) By (i) and the hypothesis \( (P \rightarrow_1) \) we set \( P = (\nu x y)(\pi v.R'_i \mid R_j = y(z).R_j) \)

(iii) We show a reduction similar to that of the case [Com] in soundness for \( \llbracket P \rrbracket_{\pi \tau} \) to show that:

\[
S \in \llbracket [P]_{\pi \tau} \rrbracket,
S \triangleq \exists x, y. (\llbracket \pi x y \rrbracket \mid \forall \epsilon((\text{rcv}(y, v) \otimes \{x : y\}) \rightarrow \llbracket P \rrbracket_{\pi \tau}) \mid \text{rcv}(y, v) \mid \llbracket P'' \{v/z\} \rrbracket_{\pi \tau})
\]

and that \( S \rightarrow_{\tau} S' \) such that

\[
S' \triangleq \exists x, y. (\llbracket \pi x y \rrbracket \mid \llbracket P' \rrbracket_{\pi \tau} \mid \llbracket P'' \{v/z\} \rrbracket_{\pi \tau})
\]

(iv) Set \( Q = (\nu x y)(P' \mid P'' \{v/z\}) \), and by the reduction rules of \( \pi \tau \) we have that \( P \rightarrow_{\pi} Q \), and conclude: \( \llbracket Q \rrbracket_{\pi \tau} \approx S' \)

Subcase \( R_i = \pi.v.R'_i \) and \( R_j = y(z).R'_j \); Analogous to the input/output case; the only consideration is the replicated input process, which is treated as in case [Com] of the completeness result.

Subcase \( R_i = \langle l_i.R'_i \rangle \) and \( R_j = x \triangleright \langle l_i : P_i \rangle \): is similar to the previous cases. We again appeal to bisimilarity, since \( S' \approx \llbracket Q \rrbracket_{\pi \tau} \), then again because there are blocked “garbage” processes (i.e., processes that will not be able to execute).
B Appendix to Section 5

B.1 Appendix to Section 5.1

Proposition 5.2 (On Inhabited Recursive Types). Statement on Page 25

Proof. We prove each one of the numerals:

(1) Follows directly from analyzing Fig. 2. Notice that in $\mu a.T$, we must have $T \neq \text{bool}$, $T \neq \text{end}$ and $T \neq a$, for the recursive type to make sense.

(2) The proof follows straightforwardly from the analysis of Fig. 2. In particular, we need to check whether the type variable $a$ appears or not in place of type $\text{end}$ in pretype $p$.

(3) The proof proceeds by contradiction. We assume $\Gamma, x : \mu a.q \ p \vdash P$ and $x : \mu a.q \ p$ and that $q \neq \text{un}$ and then by using structural induction on pretype $p$. The proof proceeds as follows:

\[
\Gamma, x : \mu a.q \ p \vdash P \quad \text{(Assumption)} \quad (1)
\]
\[
\mu a.q \ p \text{ is tail-recursive} \quad \text{(Assumption)} \quad (2)
\]
\[
q \neq \text{un} \quad \text{(Assumption)} \quad (3)
\]
\[
q = \text{lin} \quad \text{(By (3))} \quad (4)
\]

By structural induction on $p$, we proceed to analyze the different cases:

- **Case $p = \text{!}U.U'$**:  
  \[
  T = \mu a.\text{lin}\!U.U' \quad \text{(Assumptions)} \quad (5)
  \]
  \[
  P = \pi v.Q \quad \text{(by (5))} \quad (6)
  \]
  \[
  \Gamma, x : T \vdash \pi v.Q, \Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \quad \text{(Assumption)} \quad (7)
  \]
  We now show the derivation tree:

\[
\begin{array}{c}
\Gamma_1, x : T \vdash \pi v.Q' \\
\text{(T-Out)}
\end{array}
\]

\[
\begin{array}{c}
\text{Assumption} \\
\Gamma_1, x : T \vdash \pi v.Q \\
\text{Assumption} \\
\Gamma_2 \vdash v : U \\
\Gamma_3, x : U' \vdash v.Q
\end{array}
\]

Notice that the rightmost branch will reach a circularity, as the type $T$ will appear again, thus we reach a contradiction as we will not be able to finish the derivation tree.

- **Case $p = ?U.U'$**:  
  \[
  T = \mu a.\text{lin}?U.U' \quad \text{(Assumptions)} \quad (8)
  \]
  \[
  P = x(z).Q \quad \text{(by (8))} \quad (9)
  \]
  \[
  \Gamma, x : T \vdash x(z).Q, \Gamma = \Gamma_1 \circ \Gamma_2 \quad \text{(Assumption)} \quad (10)
  \]
We now show the derivation tree:

\[
\begin{align*}
\Gamma_1', x : T &\vdash s_{\pi} Q' \\
\vdots \\
\Gamma, x : T &\vdash s_{\pi} x(z).Q \\
\end{align*}
\]

Assumption (T:IN)

Notice that the rightmost branch will reach a circularity, as the type \( T \) will appear again, thus we reach a contradiction as we will not be able to finish the derivation tree.

• **Case** \( p = \oplus \{l_i : U_i\}_{i \in I} \):

\[
\begin{align*}
T &= \mu a.\text{lin} \oplus \{l_i : U_i\}_{i \in I} & \text{(Assumptions)} \\
P &= x \triangleleft l_j, Q, j \in I & \text{(by (11))} \\
\Gamma, x : T &\vdash s_{\pi} x \triangleleft l_j, Q, j \in I, \Gamma = \Gamma_1, \Gamma_2 & \text{(Assumption)}
\end{align*}
\]

We now show the derivation tree:

\[
\begin{align*}
\Gamma_2', x : T &\vdash s_{\pi} Q' \\
\vdots \\
\Gamma, x : T &\vdash s_{\pi} x \triangleleft l_j, Q \\
\end{align*}
\]

Assumption (T:Sel)

Notice that the rightmost branch will reach a circularity, as the type \( T \) will appear again, thus we reach a contradiction as we will not be able to finish the derivation tree.

• **Case** \( p = \& \{l_i : U_i\}_{i \in I} \):

\[
\begin{align*}
T &= \mu a.\text{lin} \& \{l_i : U_i\}_{i \in I} & \text{(Assumptions)} \\
P &= x \triangleright \{l_i : Q_i\}_{i \in I} & \text{(by (14))} \\
\Gamma, x : T &\vdash s_{\pi} x \triangleright \{l_i : Q_i\}_{i \in I}, \Gamma = \Gamma_1, \Gamma_2 & \text{(Assumption)}
\end{align*}
\]

We now show the derivation tree:

\[
\begin{align*}
\forall i \in I, \Gamma_i', x : T &\vdash s_{\pi} Q_i \\
\vdots \\
\Gamma, x : T &\vdash s_{\pi} x \triangleright \{l_i : Q_i\}_{i \in I} \\
\end{align*}
\]

Assumption (T:BRA)

Notice that the rightmost branch will reach a circularity, as the type \( T \) will appear again, thus we reach a contradiction as we will not be able to finish the derivation tree.

Hence, by contradiction the thesis holds.

\[\square\]

**Proposition 5.3 (Characteristic processes of a type).** Statement on Page 26
Proof. By induction on the structure of $T$, exploiting the definition of $\langle \cdot \rangle^x$ (cf. Def. 5.1), and the typing rules (cf. Fig. 3).

**Case** $T = \text{end}$ and $T = \text{bool}$: Immediate by using Rule (T:Ntr), considering that $\text{un}(x : \text{end})$ and $\text{un}(x : \text{bool})$ are true by definition of the $\text{un}(\cdot)$ predicate.

**Case** $T = q \times S \times U$: We distinguish two cases: either $S = \text{bool}$ or $S \neq \text{bool}.$

- **Subcase** $S = \text{bool}$: Then we have:
  
  $\begin{align*}
  P & \in \langle T \rangle^x \\
  P &= x(y).Q
  \end{align*}$

  (Assumption) (1)

  (Def. 5.1) (2)

  By the inductive hypothesis (IH), we have that $\Gamma', x : U \vdash_{\pi \pi} Q$, for some $\Gamma'$. Clearly, $\Gamma'$ refers to all free variables in $Q$, including $y$. Without loss of generality, we may infer that there is a $\Gamma''$, such that $\Gamma' = \Gamma'' \circ y : \text{bool}$; by Def. 5.3 we easily obtain that $\Gamma'' = \Gamma'' \circ y : \text{bool}$. We may then conclude by using Rule (T:I/n.sc):

  $\begin{align*}
  &x : q \times \text{bool} \times U \vdash_{\pi \pi} x : q \times \text{bool} \times U \vdash_{\pi \pi} Q \\
  &\Gamma''', x : q \times S \times U \vdash_{\pi \pi} x(y).Q \quad (T:In)
  \end{align*}$

- **Subcase** $S \neq \text{bool}$:

  $\begin{align*}
  P & \in \langle T \rangle^x \\
  P &= x(y).Q
  \end{align*}$

  (Assumption) (1)

  (By Def. 5.1) (2)

  By IH and the fact that $S \neq \text{bool}$, we may infer that there are $\Gamma_1', \Gamma_2'$ such that $\Gamma_1', y : S \vdash_{\pi \pi} Q$ and $\Gamma_2', x : U \vdash_{\pi \pi} R$. Lastly, we may then conclude by using Rules (T:Par) and (T:In):

  $\begin{align*}
  &x : q \times S \times U \vdash_{\pi \pi} x : q \times S \times U \vdash_{\pi \pi} Q \\
  &\Gamma_1', y : S \vdash_{\pi \pi} Q \\
  &\Gamma_2', x : U \vdash_{\pi \pi} R \\
  &\Gamma_1'', x : q \times S \times U \vdash_{\pi \pi} \pi y. Q \\
  &\Gamma_1'' \circ \Gamma_2', x : q \times S \times U \vdash_{\pi \pi} x(y).Q \quad (T:Par) \quad (T:In)
  \end{align*}$

**Case** $T = q \times S \times U$:

We consider two cases: either $S = \text{bool}$ or $S \neq \text{bool}$.

- **Subcase** $S = \text{bool}$:

  $\begin{align*}
  P & \in \langle T \rangle^x \\
  P &= \pi y. Q
  \end{align*}$

  (Assumption) (1)

  (By Def. 5.1) (2)

  By IH, we have that $\Gamma', x : U \vdash_{\pi \pi} Q$, for some $\Gamma'$. We may then conclude by using Rule (T:Out):

  $\begin{align*}
  x : q \times \text{bool} \times U \vdash_{\pi \pi} x : q \times \text{bool} \times U \vdash_{\pi \pi} y : \text{bool} \vdash_{\pi \pi} y : \text{bool} \\
  \Gamma'', y : \text{bool} \vdash_{\pi \pi} y \times S \times U \vdash_{\pi \pi} \pi y. Q \quad (T:Out)
  \end{align*}$
• **Subcase** $S \neq \text{bool}$:

$P \in [T]^x$ \hspace{1cm} (Assumption) \hspace{1cm} (1)

$P = \pi y.(Q \mid R)$ \hspace{1cm} (By Def. 5.1) \hspace{1cm} (2)

By IH and Def. 5.1 we may infer that there are $\Gamma'_1, \Gamma'_2, z : S$ such that $\Gamma'_1, z : S \vdash_{\pi} Q$ and $\Gamma'_2, x : U \vdash_{\pi} R$. Lastly, we may then conclude by using Rules (T:Par) and (T:Out):

$\Gamma'_1, z : S \vdash_{\pi} Q \quad \Gamma'_2, x : U \vdash_{\pi} R$ \hspace{1cm} (T:Par)

$\Gamma'_1, z : S \circ \Gamma'_2, x : Q \mid R \vdash_{\pi} \varpi.(Q \mid R)$ \hspace{1cm} (T:Out)

(3)

**Case** $T = q \oplus \{ l_i : S_i \}_{i \in I}$:

$P \in [T]^x$ \hspace{1cm} (Assumption) \hspace{1cm} (1)

$P = x \oplus l_j.Q$ \hspace{1cm} (Def. 5.1) \hspace{1cm} (2)

$j \in I$ \hspace{1cm} (Def. 5.1) \hspace{1cm} (3)

By IH, there exists $\Gamma'$ such that $\Gamma', x : T_j \vdash_{\pi} Q$, for some $j \in I$. We may then conclude by using Rule (T:Sel):

$\Gamma', x : Q \oplus \{ l_i : S_i \}_{i \in I} \vdash_{\pi} \varpi.(Q \mid R)$ \hspace{1cm} (T:Sel)

(4)

**Case** $T = \& \{ l_i : S_i \}_{i \in I}$:

$P \in [T]^x$ \hspace{1cm} (Assumption) \hspace{1cm} (1)

$P = x \& \{ l_i : Q_i \}_{i \in I}$ \hspace{1cm} (Def. 5.1) \hspace{1cm} (2)

$\forall i \in I. Q_i \in [S_i]^x$ \hspace{1cm} (Def. 5.1) \hspace{1cm} (3)

By IH there exists $\Gamma'$ such that $\Gamma', x : T_i \vdash_{\pi} Q_i$ for every $i \in I$. We may then conclude by using Rule (T:bra):

$\Gamma', x : Q \& \{ l_i : T_i \}_{i \in I} \vdash_{\pi} \varpi.(Q \mid R)$ \hspace{1cm} (T:bra)

(4)

**Case** $T = \mu a. S$: We will proceed using a direct proof with a case analysis when required. By Prop. 5.2(1) we have that $T = \mu a. q p$. Thus we apply a case analysis on $q$:

• **Subcase** $q = \text{lin}$: We proceed using a direct proof with a case analysis on the possible characteristics of the type, as stated by Prop. 5.2(2):
– $T$ is not tail recursive: We proceed by a direct proof as follows:

\[
\begin{align*}
T &= \mu a.\text{lin } p \quad \text{(Assumption)} \quad (1) \\
T &\text{ is not tail recursive} \quad \text{(Assumption)} \quad (2) \\
\sem{T}^x &= \sem{\mu a.\text{lin } p}^x = \sem{\text{lin } p}^x = \sem{p} \quad \text{(Def. 5.1)} \quad (3)
\end{align*}
\]

We apply structural induction with a case analysis on the structure of $p$ as follows (we only show the input case, all the other cases proceed as the cases above):

- **Subcase** $p = U . U'$: We distinguish two cases: either $U = \text{bool}$ or $U \neq \text{bool}$.

  - **Subsubcase** $U = \text{bool}$: Then we have:

    \[
    \begin{align*}
    P &\in \sem{\mu a.\text{lin} ? U . U'}^x \quad \text{(Assumption)} \quad (4) \\
P &= x(y) . Q \quad \text{(By Def. 5.1)} \quad (5)
    \end{align*}
    \]

    By IH and the fact that $T$ is not tail-recursive, we have that $\Gamma', x : U . U' \{T/a\} \vdash_{\pi} Q$, for some $\Gamma'$. Clearly, $\Gamma'$ refers to all free variables in $Q$, including $y$. Without loss of generality, we may infer that there is a $\Gamma''$, such that $\Gamma' = \Gamma'' \circ y : \text{bool}$; by Def. 3.3, we easily obtain that $\Gamma' = \Gamma'' \circ y : \text{bool}$. We may then conclude by using Rule (T:In):

    \[
    \begin{array}{l}
    \Gamma'' , x : \mu a.\text{lin} ? U . U' \vdash_{\pi} x(y) . Q \\
    \hline
    \Gamma'' , y : \text{bool} \circ x : U . U' \{T/a\} \vdash_{\pi} Q \\
    \hline
    x : \text{lin} \text{bool} . U . U' \vdash_{\pi} x : \text{lin} \text{bool} . U . U' \{T/a\}\{T/a\} \vdash_{\pi} Q \quad \text{(T:In)}
    \end{array}
    \]

  - **Subsubcase** $U \neq \text{bool}$:

    \[
    \begin{align*}
    P &\in \sem{\mu a.\text{lin} ? U . U'}^x \quad \text{(Assumption)} \quad (1) \\
P &= x(y) . (Q | R) \quad \text{(By Def. 5.1)} \quad (2)
    \end{align*}
    \]

    By IH, the fact that $U \neq \text{bool}$ and since $T$ is not tail-recursive, we may infer that there are $\Gamma'_1, \Gamma'_2$ such that $\Gamma'_1, y : U . U' \{T/a\} \vdash_{\pi} Q$ and $\Gamma'_2, x : U . U' \{T/a\} \vdash_{\pi} R$. Lastly, we may then conclude by using Rules (T:Par) and (T:In):

    \[
    \begin{array}{l}
    \hline
    \Gamma'_1, \Gamma'_2, x : U . U' \{T/a\} \vdash_{\pi} (Q | R) \\
    \hline
    \Gamma'_1 \circ \Gamma'_2, x : \mu a.\text{lin} ? U . U' \vdash_{\pi} x(y) . (Q | R) \quad \text{(T:In)}
    \end{array}
    \]

    Above, $D$ is the sequent $x : \mu a.\text{lin} ? U . U' \vdash_{\pi} x : \mu a.\text{lin} ? U . U' \{T/a\}$, which concludes by Rule (T:Var).

– $T$ is tail-recursive: This case does not apply, as shown by Prop. 5.2.3.

- **Subcase** $q = \text{un}$: We proceed using a direct proof as follows:

  \[
  \begin{align*}
  T &= \mu a.\text{un } p \quad \text{(Assumption)} \quad (1) \\
  \sem{T}^x &= \sem{\mu a.\text{un } p}^x = \sem{\text{un } p}^x = \sem{p} \quad \text{(Def. 5.1)} \quad (2)
  \end{align*}
  \]

  We apply structural induction with a case analysis on the structure of $p$ as follows (we only show the input case, all the other cases proceed as the cases above):
- **Subsubcase** \( p = ?U.U' \): We distinguish two cases: either \( U = \text{bool} \) or \( U \neq \text{bool} \).
  - **Subsubcase** \( U = \text{bool} \): Then we have:
    
    \[
    P \in \{ \mu a.\text{un}?U.U' \}^x
    \]
    (Assumption) \hspace{1cm} (3)

    \[
    P = x(y).Q
    \]
    (Def. 5.1) \hspace{1cm} (4)

    By IH and the fact that \( q = \text{un} \), we have that \( \Gamma', x:U'{T/a} \vdash_{\pi} Q \), for some \( \Gamma' \). Clearly, \( \Gamma' \) refers to all free variables in \( Q \), including \( y \). Without loss of generality, we may infer that there is a \( \Gamma'' \), such that \( \Gamma' = \Gamma'' \circ y: \text{bool} \). We may then conclude by using Rule (T:In):

    \[
    \frac{x: \text{un}?U.U' \vdash_{\pi} x: \text{un}?U.U' \{ T/a \}}{
    \frac{\Gamma', x: \mu a.\text{un}?U.U' \vdash_{\pi} x(y).Q}{(T:In)}}
    \]

    \[
    \begin{array}{l}
    \text{IH} \\
    \end{array}
    \]

  - **Subsubcase** \( U \neq \text{bool} \):
    
    \[
    P \in \{ \mu a.\text{un}?U.U' \}^x
    \]
    (Assumption) \hspace{1cm} (1)

    \[
    P = x(y).(Q | R)
    \]
    (By Def. 5.1) \hspace{1cm} (2)

    By IH, the fact that \( U \neq \text{bool} \) and since \( q = \text{un} \), we may infer that there are \( \Gamma'_1, \Gamma'_2 \) such that \( \Gamma'_1, y:U \{ T/a \} \vdash_{\pi} Q \) and \( \Gamma'_2, x:U'{T/a} \vdash_{\pi} R \). Lastly, we may then conclude by using Rules (T:Par) and (T:In):

    \[
    \frac{\Gamma'_1, y:U \{ T/a \} \vdash_{\pi} Q \hspace{1cm} \Gamma'_2, x:U'{T/a} \vdash_{\pi} R}{\Gamma'_1 \circ \Gamma'_2, x: \mu a.\text{un}?U.U' \vdash_{\pi} x(y).(Q | R)}
    \]
    (T:In) \hspace{1cm} (3)

    Above, \( D \) is the sequent \( x: \mu a.\text{un}?U.U' \vdash_{\pi} x: \mu a.\text{un}?U.U' \{ T/a \} \), which concludes by Rule (T:Var).

\[
\square
\]

**Proposition 5.6 (Example).** Statement on Page 27.

**Proof.** The proof proceeds by appealing to Def. 5.5 and proving that both processes indeed fulfill the mentioned characteristics (i.e., (i) they type under the same environment (ii) there exists some \( \{ \Delta \}^q \) such that \( \{ (\nu x y)(B_1 | \{ \Delta \}^q) \}_{\pi} \approx i \{ (\nu x y)(B_2 | \{ \Delta \}^q) \}_{\pi} \)).

For (i) the proof is trivial, since we state that both processes \( B_1 \) and \( B_2 \) share the same type and thus they are typed by the same environment (Eqs. 5.7):

\[
\Delta = b_1 : T_1 \cdot b_2 : T_2
\]

For (ii) we proceed in several steps:
1. First, we define a process $\{\Delta\}^{c_1}$. In this case, we pick $\overline{y} = c_1$ and $\Delta = \{b_1 : T_1\}$ so we define $\{\Delta\}^{c_1}$ as follows, where $F_{c_1}$ is assumed to implement the protocol to finish the sell from the seller (i.e., the complementary process to $F_{b_1}$):

$$\{\Delta\}^{c_1} = c_1(w_3).rcv.price.F_{c_1}$$

2. After having defined $\{\Delta\}^{c_1}$ we show the composed processes as follows:

$$P_1 = (\nu b_1 c_1) (B_1 \mid \{\Delta\}^{c_1}) \quad (4)$$
$$P_2 = (\nu b_1 c_1) (B_2 \mid \{\Delta\}^{c_1}) \quad (5)$$

We show the translations of the composed processes:

$$\llbracket P_1 \rrbracket_{\pi} = \exists b_1 c_1. (\{b_1 : c_1\} \parallel \text{snd}(b_1, book1) \parallel \forall z (rcv(z, book1) \otimes \{b_1 : z\} \rightarrow \text{snd}(b_2, book2) \parallel \forall w (rcv(w, book2) \otimes \{b_2 : w\} \rightarrow \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1 : b_1\} \rightarrow rcv(b_1, w_1) \parallel \forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2 : b_2\} \rightarrow rcv(b_2, w_2) \parallel \llbracket F_{b_1} \cdot F_{b_2} \rrbracket_{\pi}))) \parallel \llbracket \{\Delta\}^{c_1} \rrbracket_{\pi})$$

$$\llbracket P_2 \rrbracket_{\pi} = \exists b_1 c_1. (\{b_1 : c_1\} \parallel \text{snd}(b_1, book1) \parallel \forall z (rcv(z, book1) \otimes \{b_1 : z\} \rightarrow \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1 : b_1\} \rightarrow rcv(b_1, w_1) \parallel \llbracket F_{b_1} \rrbracket_{\pi}))) \parallel \forall w (rcv(w, book2) \otimes \{b_2 : w\} \rightarrow \forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2 : b_2\} \rightarrow rcv(b_2, w_2) \parallel \llbracket F_{b_2} \rrbracket_{\pi}) \parallel \llbracket \{\Delta\}^{c_1} \rrbracket_{\pi})$$

3. We prove that $\llbracket P_1 \rrbracket_{\pi} \approx_i \llbracket P_2 \rrbracket_{\pi}$. This is done by building a bisimulation relation $R$ according to Def.3.15 To do so, we show all the reductions for $\llbracket P_1 \rrbracket_{\pi}$ and $\llbracket P_2 \rrbracket_{\pi}$ respectively as follows. First we see the possible reductions for

$$\llbracket P_1 \rrbracket_{\pi} \overset{\pi}{\rightarrow} S_1^{(1)} \overset{\pi}{\rightarrow} S_1^{(2)} \overset{\text{snd}(b_2, book2)}{\rightarrow} S_1^{(3)} \not\rightarrow_1 \text{ where}$$

$$S_1^{(1)} = \exists b_1 c_1. (\{b_1 : c_1\} \parallel \forall z (rcv(z, book1) \otimes \{b_1 : z\} \rightarrow \text{snd}(b_2, book2) \parallel \forall w (rcv(w, book2) \otimes \{b_2 : w\} \rightarrow \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1 : b_1\} \rightarrow rcv(b_1, w_1) \parallel \forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2 : b_2\} \rightarrow rcv(b_2, w_2) \parallel \llbracket F_{b_1} \cdot F_{b_2} \rrbracket_{\pi})))$$

$$S_1^{(2)} = \exists b_1 c_1. (\{b_1 : c_1\} \parallel \text{snd}(b_2, book2) \parallel \forall w (rcv(w, book2) \otimes \{b_2 : w\} \rightarrow \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1 : b_1\} \rightarrow rcv(b_1, w_1) \parallel \forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2 : b_2\} \rightarrow rcv(b_2, w_2) \parallel \llbracket F_{b_1} \cdot F_{b_2} \rrbracket_{\pi})))$$

$$S_1^{(3)} = \exists b_1 c_1. (\{b_1 : c_1\} \parallel \forall w (rcv(w, book2) \otimes \{b_2 : w\} \rightarrow \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1 : b_1\} \rightarrow rcv(b_1, w_1) \parallel \forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2 : b_2\} \rightarrow rcv(b_2, w_2) \parallel \llbracket F_{b_1} \cdot F_{b_2} \rrbracket_{\pi})))$$

63
Now we check the all possible reductions for $\llbracket P_2 \rrbracket_{\pi}$:

\[
\llbracket P_2 \rrbracket_{\pi} \xrightarrow{\tau} S_2^{(1)} \xrightarrow{\tau} S_2^{(2)} \xrightarrow{\tau} S_2^{(3)} \xrightarrow{\tau} S_2^{(4)} \xrightarrow{\text{snd}(b_2,\text{book}2)} S_2^{(5)} \not\xrightarrow{1}
\]
\[
\llbracket P_2 \rrbracket_{\pi} \xrightarrow{\tau} S_2^{(1)} \xrightarrow{\tau} S_2^{(2)} \xrightarrow{\tau} S_2^{(3)} \xrightarrow{\text{snd}(b_2,\text{book}2)} S_2^{(4)} \xrightarrow{\tau} S_2^{(6)} \not\xrightarrow{1}
\]
\[
\llbracket P_2 \rrbracket_{\pi} \xrightarrow{\tau} S_2^{(1)} \xrightarrow{\tau} S_2^{(2)} \xrightarrow{\tau} S_2^{(3)} \xrightarrow{\text{snd}(b_2,\text{book}2)} S_2^{(4)} \xrightarrow{\tau} S_2^{(7)} \not\xrightarrow{1}
\]
\[
\llbracket P_2 \rrbracket_{\pi} \xrightarrow{\tau} S_2^{(1)} \xrightarrow{\tau} S_2^{(2)} \xrightarrow{\tau} S_2^{(3)} \xrightarrow{\text{snd}(b_2,\text{book}2)} S_2^{(4)} \xrightarrow{\tau} S_2^{(7)} \not\xrightarrow{1}
\]
\[
\llbracket P_2 \rrbracket_{\pi} \xrightarrow{\tau} S_2^{(1)} \xrightarrow{\tau} S_2^{(2)} \xrightarrow{\tau} S_2^{(3)} \xrightarrow{\text{snd}(b_2,\text{book}2)} S_2^{(4)} \xrightarrow{\tau} S_2^{(9)} \not\xrightarrow{1}
\]
where:

\[S_2^{(1)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \forall z (\text{rcv}(z, \text{book}1) \otimes \{b_1: z\}) \rightarrow \]
\[
\\forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1: b_1\}) \rightarrow \text{rcv}(b_1, w_1) \parallel \llbracket F_1 \rrbracket_{\pi}
\]
\[
\| \text{snd}(b_2, \text{book}2) \| \ \forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[S_2^{(2)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1: b_1\}) \rightarrow \text{rcv}(b_1, w_1) \parallel \llbracket F_1 \rrbracket_{\pi}
\]
\[
\| \text{snd}(b_2, \text{book}2) \| \ \forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[S_2^{(3)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \text{rcv}(b_1, \text{price}1) \parallel \llbracket F_1 \rrbracket_{\pi} \| \text{snd}(b_2, \text{book}2) \|
\]
\[
\\forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[
\| \text{snd}(b_2, \text{book}2) \| \ \forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[S_2^{(4)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \text{rcv}(b_1, \text{price}1) \parallel \llbracket F_1 \rrbracket_{\pi} \| \text{snd}(b_2, \text{book}2) \|
\]
\[
\\forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[S_2^{(5)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \text{rcv}(b_1, \text{price}1) \parallel \llbracket F_1 \rrbracket_{\pi} \| \forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[S_2^{(6)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \llbracket F_1 \rrbracket_{\pi} \| \forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
\[S_2^{(7)} = \exists b_1 c_1, (!\{b_1: c_1\}) \parallel \forall z_1 w_1 (\text{snd}(z_1, w_1) \otimes \{z_1: b_1\}) \rightarrow \text{rcv}(b_1, w_1) \parallel \llbracket F_1 \rrbracket_{\pi} \| \forall w (\text{rcv}(w, \text{book}2) \otimes \{b_2: w\}) \rightarrow \]
\[
\\forall z_2 w_2 (\text{snd}(z_2, w_2) \otimes \{z_2: b_2\}) \rightarrow \text{rcv}(b_2, w_2) \parallel \llbracket F_2 \rrbracket_{\pi}
\]
Since with

By induction on the structure of

Proposition B.1 (Name invariance for

B.2.1 Correctness for the encoding

B.2 Appendix to Section 5.2

(a) /The trace for

K

is a bisimulation containing

π

CH

(8)

CH

(Def. 5.16(b)), and a variable in

π

CH

, resp. /Then

π

CH

, and

S

1

(9)

=∃b_1c_1, (!!b_1c_1) \land \forall z (rcv(z, book1) \otimes \{b_1\}:z) \\
\forall z_1w_1(snd(z_1, w_1) \otimes \{z_1\}:b_1) \rightarrow r(cv(b_1, w_1) \parallel \| F_{b_1\parallel z_1})
\\
\parallel w(rcv(w, book2) \otimes \{b_2\}:w) \\
\forall z_2w_2(snd(z_2, w_2) \otimes \{z_2\}:b_2) \rightarrow r(cv(b_2, w_2) \parallel \| F_{b_2\parallel z_2})
\\
\parallel r(cv(b_1, book1) \parallel snd(c_1, price1) \parallel \forall u(r(cv(u, price1) \otimes \{c_1\}:u) \rightarrow \| F_{c_1\parallel z_2})
\\
S_{2}^{9} = ∃b_1c_1, (!!b_1c_1) \land \forall z (rcv(z, book1) \otimes \{b_1\}:z) \\
\forall z_1w_1(snd(z_1, w_1) \otimes \{z_1\}:b_1) \rightarrow r(cv(b_1, w_1) \parallel \| F_{b_1\parallel z_1})
\\
\parallel w(rcv(w, book2) \otimes \{b_2\}:w) \\
\forall z_2w_2(snd(z_2, w_2) \otimes \{z_2\}:b_2) \rightarrow r(cv(b_2, w_2) \parallel \| F_{b_2\parallel z_2}) \parallel \| ([\Delta] (c_1)_{\parallel z_1})

4. Lastly, we build a bisimulation relation \( \mathcal{R} \) to prove that \( B_1 \simeq_{\pi}^b B_2 \). We consider the following:

(a) The trace for \( \| P_1 \|_{\pi} \):

\[
\| P_1 \|_{\pi} \xrightarrow{\tau} S_{1}^{(1)} \xrightarrow{\tau} S_{1}^{(2)} \xrightarrow{snd(b_2, book2)} S_{1}^{(3)} \not\rightarrow_{1}
\]

shows that it is necessary to include the following pairs in \( \mathcal{R} \):

\[
(\| P_1 \|_{\pi}, \| P_2 \|_{\pi}), (S_{1}^{(1)}, S_{2}^{(1)}), (S_{1}^{(2)}, S_{2}^{(1)}), (S_{1}^{(3)}, S_{2}^{(5)})
\]

From the second process, we take in the following pairs:

\[
(S_{1}^{(1)}, S_{2}^{(1)}), (S_{1}^{(2)}, S_{2}^{(2)}), (S_{1}^{(1)}, S_{2}^{(3)}), (S_{1}^{(4)}, S_{2}^{(4)}), (S_{1}^{(3)}, S_{2}^{(6)}), (S_{1}^{(3)}, S_{2}^{(7)}), (S_{1}^{(3)}, S_{2}^{(8)}), (S_{1}^{(3)}, S_{2}^{(9)})
\]

Thus, we conclude that the bisimulation relation \( \mathcal{R} \) should be as follows:

\[
\mathcal{R} = \{(\| P_1 \|_{\pi}, \| P_2 \|_{\pi}), (S_{1}^{(1)}, S_{2}^{(1)}), (S_{1}^{(2)}, S_{2}^{(1)}), (S_{1}^{(3)}, S_{2}^{(5)}),
(S_{1}^{(1)}, S_{2}^{(1)}), (S_{1}^{(2)}, S_{2}^{(2)}), (S_{1}^{(1)}, S_{2}^{(3)}), (S_{1}^{(4)}, S_{2}^{(4)}),
(S_{1}^{(3)}, S_{2}^{(6)}), (S_{1}^{(3)}, S_{2}^{(7)}), (S_{1}^{(3)}, S_{2}^{(8)}), (S_{1}^{(3)}, S_{2}^{(9)})\}
\]

Since \( \mathcal{R} \) is a bisimulation containing \( \| P_1 \|_{\pi}, \| P_2 \|_{\pi} \), we have \( P_1 \simeq_{\pi} P_2 \) and thus \( B_1 \simeq_{\pi}^b B_2 \).

B.2 Appendix to Section 5.2

B.2.1 Correctness for the encoding

Proposition B.1 (Name invariance for \( \| P \|_{\pi} \)). Let \( P, \sigma, \) and \( x \) be a typable \( \pi_{\text{CH}} \) process, a substitution satisfying the renaming policy for \( \| P \|_{\pi} \) (Def. 5.16(b)), and a variable in \( \pi_{\text{CH}} \), resp. Then \( \| P\sigma \|_{\pi_{\text{CH}}} = \| P \|_{\pi_{\text{CH}}} \sigma' \), with \( \varphi \|_{\pi_{\text{CH}}} (\sigma(x)) = \sigma'(\varphi \|_{\pi_{\text{CH}}} (x)) \) and \( \sigma = \sigma' \).

Proof. By induction on the structure of \( P \). The proof proceeds is analogous to the proof of Thm.4.6
Proposition B.2 (Compositionality of $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$). Let $P$ and $E[\cdot]$ be a typable $\pi_{\text{cc}}$ process without forwarding operators and an $\pi_{\text{cc}}$ evaluation context (for reference see Def. 4.7), respectively. Then we have: $\llbracket E[P] \rrbracket_{\pi_{\text{cc}}} = \llbracket E \rrbracket_{\pi_{\text{cc}}} \llbracket P \rrbracket_{\pi_{\text{cc}}}$.

Proof. By induction on the structure of $P$. The proof proceeds is analogous to the proof of Thm. 4.8.

Before proving operational correspondence, we prove an auxiliary result corresponding to the forwarding operator (the main novelty in mapping $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$, cf. Def. 5.16):

Theorem B.3 (Operational correspondence of the forward operator). Let $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$ be the translation in Def. 5.16. Also, let $P, P'$ be well-typed $\pi_{\text{cc}}$ programs, $S$ be an $\lambda$cc process and $x, y$ be names. Then:

1. Soundness: If $P \equiv_{x} (\nu \nu)([x \leftrightarrow y] \mid P') \rightarrow_{\pi_{\text{cc}}} P' \{y/x\}$ then
   \[ \llbracket P \rrbracket_{\pi_{\text{cc}}} (x)(1_{x=y}) \rightarrow_{1} S, \text{ such that } S \{y/x\} \approx_{1} \llbracket P' \{y/x\} \rrbracket_{\pi_{\text{cc}}}. \]

2. Completeness: If $\llbracket P \rrbracket_{\pi_{\text{cc}}} (x)(1_{x=y}) \rightarrow_{1} S$ then $P \equiv_{\pi_{\text{cc}}} (\nu \nu)([x \leftrightarrow z] \mid P') \rightarrow_{\pi_{\text{cc}}} P' \{z/x\}$ and $S \approx_{1} \llbracket P' \{z/x\} \rrbracket_{\pi_{\text{cc}}}$.

Proof. We prove each item separately:

1. Soundness: The proof proceeds straightforwardly by induction on the reduction of $P$. The only case that applies is that of Rule $[\text{Fwd}_{\pi_{\text{cc}}}]$, thus $P = (\nu \nu)([x \leftrightarrow y] \mid P')$. Then by the definition of $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$ (cf. Def. 5.16)
   \[ \llbracket P \rrbracket_{\pi_{\text{cc}}} = \exists x. !x \approx y \mid \llbracket P' \rrbracket_{\pi_{\text{cc}}} = \exists x. !x \approx y \mid \llbracket P' \rrbracket_{\pi_{\text{cc}}}. \]
   and by the operational semantics of $\lambda$cc (cf. 5), we have that
   \[ \exists x. !x \approx y \mid !x \approx y \mid \llbracket P' \rrbracket_{\pi_{\text{cc}}} \rightarrow_{1} \llbracket P' \rrbracket_{\pi_{\text{cc}}}. \]
   which is the process that we wanted to find, since $\llbracket P' \rrbracket_{\pi_{\text{cc}}} \{y/x\} = \llbracket P' \{y/x\} \rrbracket_{\pi_{\text{cc}}}$.

2. Completeness: The proof proceeds by induction on the transition. Note that by Fig. 5 and the structure of $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$ (cf. Def. 5.16), the only interesting rule is Rule $([\text{CExt}])$. This means that constraint $!x = y$ is being sent to the store with a restricted variable $x$. Thus, $\llbracket P \rrbracket_{\pi_{\text{cc}}} = \exists x. !x \approx y \mid R$. By the definition of $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$ the only process that could generate such constraint is $(\nu \nu)([x \leftrightarrow y] \mid P')$, and therefore, $P \equiv_{\pi_{\text{cc}}} (\nu \nu)([x \leftrightarrow y] \mid P')$. All the other cases proceed by applying the induction hypothesis.

Theorem B.4 (Operational Correspondence for $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$). Let $\llbracket \cdot \rrbracket_{\pi_{\text{cc}}}$ be the translation in Def. 5.16. Also, let $P, Q$ be well-typed $\pi_{\text{cc}}$ programs, $R, S$ be $\lambda$cc processes and $x, y$ be names. Then:

1. Soundness: If $P \rightarrow_{\pi_{\text{cc}}} Q$ then either:
   a. $\llbracket P \rrbracket_{\pi_{\text{cc}}} \rightarrow_{1} R$, such that $R \approx_{1} \llbracket Q \rrbracket_{\pi_{\text{cc}}}$.
   b. $(\text{or}) \llbracket P \rrbracket_{\pi_{\text{cc}}} \equiv_{1} S' \rightarrow_{1} R'$, for some $R', S'$, $R$ such that $R \approx_{1} \llbracket Q \rrbracket_{\pi_{\text{cc}}}$.

2. Completeness: If $\llbracket P \rrbracket_{\pi_{\text{cc}}} \rightarrow_{1} S$ then either:
a. $P \rightarrow_{\pi_{\text{cc}}} Q$, for some $Q$ and $\llbracket Q \rrbracket_{\pi_{\text{cc}}} \approx S$.

b. (or) $S \in \llbracket \llbracket P \rrbracket_{\pi_{\text{cc}}} \rrbracket$ and, for some $S'$ and $Q$, we have $S \rightarrow_{\pi_{\text{cc}}} S'$, $P \rightarrow_{\pi_{\text{cc}}} Q$, and $\llbracket Q \rrbracket_{\pi_{\text{cc}}} \approx S'$.

**Proof.** For both cases the proof proceeds as in Thm. 4.12.

The combination of Thm. B.3 and Thm. B.4 gives us operational correspondence in the sense of Def. 4.2.

### B.2.2 Proof of Lemma 5.20

**Lemma 5.20 (Live Processes).** Statement on Page 33.

**Proof.** We prove the two cases of the biconditional.

**Case $\Rightarrow$** We proceed directly by using the assumption live$_{\pi_{\text{cc}}}(M)$:

$$M \equiv_1 (\exists \bar{x}. (S \parallel R))$$  \hspace{1cm} (Assumption)

By a case analysis on $S$:

**Case $S = \lfloor x(y).P \rrbracket_{\pi_{\text{cc}}}$ or $S = \lfloor x \triangleright \{ l_i : P_i \}_{i \in I} \rrbracket_{\pi_{\text{cc}}}$**

Straightforward from the definition of the encoding.

**Case $S = \lfloor \text{snd}(x,v) \rrbracket$**

$$R \equiv_1 \forall \epsilon (\text{rcv}(x,v) \rightarrow S) \parallel R$$  \hspace{1cm} (Def. of $\llbracket \rrbracket_{\pi_{\text{cc}}}$)

$S \parallel R$ contains the encoding of an output, and thus live$_{\pi_{\text{cc}}}(P)$ holds.

**Case $S = \lfloor \text{sel}(x,l_i) \rrbracket$**

This case is analogous to the previous one.

**Case $\Leftarrow$** This case proceeds straightforwardly by appealing to the definition of $\llbracket \rrbracket_{\pi_{\text{cc}}}$.

$$\text{live}_{\pi_{\text{cc}}}(P)$$  \hspace{1cm} (Assumption) \hspace{1cm} (1)

$$P \equiv_{\pi_{\text{cc}}} (\nu \bar{u}) \cdot P \parallel R$$  \hspace{1cm} By (1) \hspace{1cm} (2)

We distinguish several cases, depending on the shape of prefix $\pi$. We content ourselves by detailing two cases:

**Case $\pi = x(y)$:**

$$P \equiv_{\pi_{\text{cc}}} (\nu \bar{u}) (x(y)).Q \parallel R$$  \hspace{1cm} (Assumption) \hspace{1cm} (1)

$$\llbracket P \rrbracket_{\pi_{\text{cc}}} = (\exists \bar{u}. \forall y (\text{snd}(x, y) \rightarrow \text{rcv}(x,y) \parallel \llbracket Q \rrbracket_{\pi_{\text{cc}}} \parallel \llbracket R \rrbracket_{\pi_{\text{cc}}}))$$  \hspace{1cm} (Def. 5.16) \hspace{1cm} (2)

$$\text{live}_{1cc}(\llbracket P \rrbracket_{\pi_{\text{cc}}})$$  \hspace{1cm} (By (2) and Def. 5.18) \hspace{1cm} (3)

**Case $\pi = \bar{x} \cdot v$:**

$$P \equiv_{\pi_{\text{cc}}} (\nu \bar{u}) (\bar{x} \cdot v).Q \parallel R$$  \hspace{1cm} (Assumption) \hspace{1cm} (1)

$$\llbracket P \rrbracket_{\pi_{\text{cc}}} = \text{snd}(x,v) \parallel \forall \epsilon (\text{rcv}(x,v) \rightarrow \llbracket Q \rrbracket_{\pi_{\text{cc}}} \parallel \llbracket R \rrbracket_{\pi_{\text{cc}}})$$  \hspace{1cm} (Def. 5.16) \hspace{1cm} (2)

$$\text{live}_{1cc}(\llbracket P \rrbracket_{\pi_{\text{cc}}})$$  \hspace{1cm} (By (2) and Def. 5.18) \hspace{1cm} (3)

All other cases proceed similarly as above. \qed
Definition C.1 (Substitution). Given terms \( \bar{t} = t_1, \ldots, t_n \) and process variables \( \bar{x} = x_1, \ldots, x_n \), the application of a substitution to a constraint, guard and process, denoted respectively \( c[\bar{t}/\bar{x}] \), \( G[\bar{t}/\bar{x}] \), and \( P[\bar{t}/\bar{x}] \), is inductively defined on the structure of constraints, guards and process as in Fig. 18.

Lemma 6.1 (Subject Congruence). Statement on Page 37.

Proof. The proof proceeds by induction on the depth of the premise \( P \equiv Q \). Since congruences are symmetric, we need to prove for both \( P \equiv Q \) and \( Q \equiv P \).

Case \( P \equiv Q \): The case is trivial since it is just a renaming of bound variables.

Case \( P \parallel \bar{t} \equiv_1 P \)

Subcase (\( \Rightarrow \)):

<table>
<thead>
<tr>
<th>Assumption</th>
<th>( P \parallel \bar{t} \equiv_1 P )</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumption</td>
<td>( \vdash_\circ P \parallel \bar{t} )</td>
<td>(2)</td>
</tr>
<tr>
<td>2, inversion on Rule (L:Par)</td>
<td>( \vdash_\circ \bar{t} )</td>
<td>(3)</td>
</tr>
<tr>
<td>2, inversion on Rule (L:Par)</td>
<td>( \vdash_\circ P )</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Figure 18: Substitution.
Subcase ($\iff$):

Assumption $P \equiv_1 P \parallel \top$ (1)

Assumption $\vdash_0 P$ (2)

By Rule (L:TELL) $\vdash_0 \top$ (3)

2,3 using Rule (L:PAR) $\vdash_0 P \parallel \top$ (4)

Case $P \parallel Q \equiv_1 Q \parallel P$

Subcase ($\Rightarrow$):

Assumption $P \parallel Q \equiv_1 Q \parallel P$ (1)

Assumption $\vdash_0 P \parallel Q$ (2)

2, inversion on Rule (L:PAR) $\vdash_0 Q$ (3)

2, inversion on Rule (L:PAR) $\vdash_0 P$ (4)

3, 4, using Rule (L:PAR) $\vdash_0 Q \parallel P$ (5)

Subcase ($\Leftarrow$):

Assumption $Q \parallel P \equiv_1 P \parallel Q$ (1)

Assumption $\vdash_0 Q \parallel P$ (2)

2, inversion on Rule (L:PAR) $\vdash_0 P$ (3)

2, inversion on Rule (L:PAR) $\vdash_0 Q$ (4)

3, 4, using Rule (L:PAR) $\vdash_0 P \parallel Q$ (5)

Case $P \parallel (Q \parallel R) \equiv_1 (P \parallel Q) \parallel R$

Subcase ($\Rightarrow$):

Assumption $P \parallel (Q \parallel R) \equiv_1 (P \parallel Q) \parallel R$ (1)

Assumption $\vdash_0 P \parallel (Q \parallel R)$ (2)

2, inversion on Rule (L:PAR) $\vdash_0 P$ (3)

2, inversion on Rule (L:PAR) $\vdash_0 (Q \parallel R)$ (4)

4, inversion on Rule (L:PAR) $\vdash_0 Q$ (5)

4, inversion on Rule (L:PAR) $\vdash_0 R$ (6)

3, 5, using Rule (L:PAR) $\vdash_0 P \parallel Q$ (7)

7, 6, using Rule (L:PAR) $\vdash_0 (P \parallel Q) \parallel R$ (8)

Subcase ($\Leftarrow$):

Assumption $(P \parallel Q) \parallel R \equiv_1 P \parallel (Q \parallel R)$ (1)

Assumption $\vdash_0 (P \parallel Q) \parallel R$ (2)

2, inversion on Rule (L:PAR) $\vdash_0 P \parallel Q$ (3)

2, inversion on Rule (L:PAR) $\vdash_0 R$ (4)

3, inversion on Rule (L:PAR) $\vdash_0 P$ (5)
3, inversion on Rule (L:PAR) \( \vdash \Box Q \) (6)
6, 4, using Rule (L:PAR) \( \vdash Q \parallel R \) (7)
5, 7, using Rule (L:PAR) \( \vdash P \parallel (Q \parallel R) \) (8)

Case \( \exists z. \overline{T} \equiv \overline{1} \overline{T} \)

Subcase (\( \Rightarrow \)):

Assumption \( \exists z. \overline{T} \equiv \overline{1} \overline{T} \) (1)
Assumption \( \vdash \exists z. \overline{T} \) (2)
2, inversion on Rule (L:EXIST) \( \vdash \overline{T} \) (3)

Subcase (\( \Leftarrow \)):

Assumption \( T \equiv \overline{1} \exists z. \overline{T} \) (1)
1, Rule (L:EXIST) \( \vdash \exists z. \overline{T} \) (2)

Case \( \exists x. \exists y. P \equiv \overline{1} \exists y. \exists x. P \)

Subcase (\( \Rightarrow \)):

Assumption \( \exists x. \exists y. P \equiv \overline{1} \exists y. \exists x. P \) (1)
Assumption \( \vdash \exists x. \exists y. P \) (2)
2, inversion on Rule (L:LOCAL) \( \vdash \exists y. P \) (3)
3, inversion on Rule (L:LOCAL) \( \vdash P \) (4)
4, formation with Rule (L:LOCAL) and \( \overline{x} \) \( \vdash \exists x. P \) (5)
5, formation with Rule (L:LOCAL) and \( \overline{y} \) \( \vdash \exists y. \exists x. P \) (6)

Subcase (\( \Leftarrow \)):

Assumption \( \exists y. \exists x. P \equiv \overline{1} \exists x. \exists y. P \) (1)
Assumption \( \vdash \exists y. \exists x. P \) (2)
2, inversion on Rule (L:LOCAL) \( \vdash \exists x. P \) (3)
3, inversion on Rule (L:LOCAL) \( \vdash P \) (4)
4, formation with Rule (L:LOCAL) and \( \overline{y} \) \( \vdash \exists y. P \) (5)
5, formation with Rule (L:LOCAL) and \( \overline{x} \) \( \vdash \exists x. \exists x. P \) (6)

Case \( !P \equiv \overline{1} P \parallel !P \)

Subcase (\( \Rightarrow \)):

Assumption \( !P \equiv \overline{1} P \parallel !P \) (1)
Assumption \( \vdash !P \) (2)
2, inversion on Rule (L:REPL) \( \vdash P \) (3)
2, 3, formation on Rule (L:PAR) \( \vdash P \parallel !P \) (4)

Subcase (\( \Leftarrow \)):

Assumption \( !P \parallel !P \equiv \overline{1} !P \) (1)
Case $\overline{e} \parallel \overline{d} \equiv_1 \overline{e}$

Subcase ($\Rightarrow$):

Assumption $\overline{e} \parallel \overline{d} \equiv_1 \overline{e}$ (1)

Assumption $\vdash_0 \overline{e} \parallel \overline{d}$ (2)

1, inversion on Rule (ScL:5) $c \otimes d \not\vdash e$ (3)

2, inversion on Rule (L:PAR) $\vdash_0 \overline{e}$ (4)

2, inversion on Rule (L:PAR) $\vdash_0 \overline{d}$ (5)

4, inversion on Rule (L:TELL) $c \in \mathcal{C}$ (6)

5, inversion on Rule (L:TELL) $d \in \mathcal{C}$ (7)

6,7 conjunction $c \otimes d \in \mathcal{C}$ (8)

3,8, entailment and deductive closure $e \in \mathcal{C}$ (9)

9, formation on Rule (L:TELL) $\vdash_0 \overline{e}$ (10)

Subcase ($\Leftarrow$):

Assumption $\overline{e} \equiv_1 \overline{e} \parallel \overline{d}$ (1)

Assumption $\vdash_0 \overline{e}$ (2)

1, inversion on Rule (ScL:5) $e \not\vdash c \otimes d$ (3)

2, inversion on Rule (L:PAR) $e \in \mathcal{C}$ (4)

3, entailment and deductive closure $c \otimes d \in \mathcal{C}$ (5)

5, inversion $c \in \mathcal{C}$ (6)

5, inversion $d \in \mathcal{C}$ (7)

6, formation on Rule (L:TELL) $\vdash_0 \overline{e}$ (8)

7, formation on Rule (L:TELL) $\vdash_0 \overline{d}$ (9)

8, 9, formation on Rule (L:PAR) $\vdash_0 \overline{e} \parallel \overline{d}$ (10)

Case $P \parallel Q \equiv_1 P' \parallel Q$

Subcase ($\Rightarrow$):

Assumption $P \parallel Q \equiv_1 P' \parallel Q$ (1)

Assumption $\vdash_0 P \parallel Q$ (2)

1, inversion on Rule (ScL:6) $P \equiv_1 P'$ (3)

2, inversion on Rule (L:PAR) $\vdash_0 P$ (4)

2, inversion on Rule (L:PAR) $\vdash_0 Q$ (5)

3,4, IH $\vdash_0 P'$ (6)

5,6, formation on Rule (L:PAR) $\vdash_0 P' \parallel Q$ (7)
**Subcase (⇐):** Analogous to the previous case.

**Case** $P \parallel \exists z. Q \equiv_1 \exists z. (P \parallel Q)$

**Subcase (⇒):**

Assumption $P \parallel \exists z. Q \equiv_1 \exists z. (P \parallel Q)$ (1)

Assumption $\vdash P \parallel \exists z. Q$ (2)

1, inversion on Rule (ScL:7) $z \notin f v(P)$ (3)

2, inversion on Rule (L:PAR) $\vdash P$ (4)

2, inversion on Rule (L:PAR) $\vdash \exists z. Q$ (5)

5, inversion on Rule (L:L/LOC) $\vdash \exists z. Q$ (6)

4,6, formation on Rule (L:PAR) $\vdash P \parallel Q$ (7)

7, formation on Rule (L:L/LOC) using $z$ $\vdash \exists z. (P \parallel Q)$ (8)

**Subcase (⇐):**

Assumption $\exists z. (P \parallel Q) \equiv_1 P \parallel \exists z. Q$ (1)

Assumption $\vdash \exists z. (P \parallel Q)$ (2)

1, inversion on Rule (ScL:7) $z \notin f v(P)$ (3)

2, inversion on Rule (L:PAR) $\vdash P \parallel Q$ (4)

4, inversion on Rule (L:PAR) $\vdash P$ (5)

4, inversion on Rule (L:PAR) $\vdash Q$ (6)

6, formation on Rule (L:PAR) $\vdash \exists z. Q$ (7)

5,7, formation on Rule (L:L/LOC) using $z$ $\vdash \exists z. (P \parallel Q)$ (8)

**Case** $\exists x. P \equiv_1 \exists x. P'$

**Subcase (⇒):**

Assumption $\exists x. P \equiv_1 \exists x. P'$ (1)

Assumption $\vdash \exists x. P$ (2)

1, inversion on Rule (ScL:8) $P \equiv_1 P'$ (3)

2, inversion on Rule (L:PAR) $\vdash P$ (4)

3, 4, IH $\vdash P'$ (5)

5, formation on Rule (L:L/LOC) $\vdash \exists x. P'$ (6)

**Subcase (⇐):** Analogous to the previous subcase.

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

**Proof.** By induction on the structure of $P$. Interesting cases are when $P = \tau$ and $P = \forall \gamma(d; e \rightarrow P')$, for some $\gamma$, $d$, $e$, and $P'$; other cases are straightforward.

- Case $P = \tau$: By the well-typedness assumption we infer that $c \in C$, which immediately implies that $c\{t/x\} \in C$ and so we conclude using (L:Tell).

---

72
• Case \( P = \forall \vec{y}(d ; e \rightarrow P') \): By the well-typedness assumption we infer that \( P' \) is well-typed, that \( \Delta; \Theta \vdash \ast e \), and that \( \vec{y} \subseteq \Theta \setminus fv(d) \). Since universal and existential quantifiers are binders, occurrences of variables in \( \vec{y} \) are not affected by \( \{t/x\} \); this in particular rules out the possibility of renaming an unrestricted variable into a restricted one. The substitution thus only affects free variables (not in \( \vec{y} \)) and the thesis follows.

\[ \square \]

**Theorem 6.2 (Type Preservation).** Statement on Page 37.

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

**Case Rule (C:Out):**

1. Assumption
2. Inversion on Rule (C:Out)
3. Inversion on Rule (C:Out)
4. Inversion on Rule (C:Out)
5. Assumption
6. Inversion on Rule (L:Tell)
7. Deductive closure, transitivity
8. Formation on Rule (L:Tell)

**Case Rule (C:SyncLoc):**

1. Assumption
2. Inversion on Rule (C:SyncLoc)
3. Inversion on Rule (C:SyncLoc)
4. Inversion on Rule (C:SyncLoc)
5. Assumption
6. Inversion on Rule (L:Par)
7. Inversion on Rule (L:Par)
8. Inversion on Rule (L:Guard)
9. Inversion on Rule (L:Abs)
10. Inversion on Rule (L:Abs)
11. Delta: \( \Theta \vdash \ast e \)
12. Inversion on Rule (L:Abs)
13. Transitivity
14. Formation on Rule (L:Tell)
15. Formulation
16. Definition of mgc

73
10, 16, Prop. C.2
\[ \vdash_\phi P[\vec{y}/\vec{x}] \] (17)

15, 17, formation on Rule (L:\PAR)
\[ \vdash_\phi P[\vec{y}/\vec{x}] \parallel \vec{f} \] (18)

18, formation on Rule (L:\LOCAL) with \( \vec{y} \)
\[ \vdash_\phi \exists \vec{y}. (P[\vec{y}/\vec{x}] \parallel \vec{f}) \] (19)

**Case Rule (C:IN):**

- **Assumption** \( \top \xrightarrow{\vec{r}} \tau \) (1)
- **Assumption** \( \vdash_\phi \top \) (2)
  - 2, inversion on Rule (L:\TELL) \( 1 \in C \) (3)
  - 1, well-formedness of labels \( c \in C \) (4)
  - 4, formation on Rule (L:\TELL) \( \vdash_\phi c \) (5)

**Case Rule (C:COMP):**

- **Assumption** \( P \parallel Q \xrightarrow{\alpha} P' \parallel Q \) (1)
  - 1, inversion on Rule (C:COMP) \( P \xrightarrow{\alpha} P' \) (2)
  - 1, inversion on Rule (C:COMP) \( ev(\alpha) \cap fv(Q) = \emptyset \) (3)
  - **Assumption** \( \vdash_\phi P' \parallel Q \) (4)
  - 4, inversion on Rule (L:\PAR) \( \vdash_\phi P \) (5)
  - 4, inversion on Rule (L:\PAR) \( \vdash_\phi Q \) (6)
  - 2, 5, IH \( \vdash_\phi P' \) (7)
  - 6, 7, formation on Rule (L:\PAR) \( \vdash_\phi P' \parallel Q \) (8)

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

- **Assumption** \( P \parallel G_1 + G_2 \xrightarrow{\alpha} Q \) (1)
  - 1, inversion on Rule (C:SUM) \( P \parallel G_1 \xrightarrow{\alpha} Q \) (2)
  - **Assumption** \( \vdash_\phi P \parallel G_1 + G_2 \) (3)
  - 3, inversion on Rule (L:\PAR) \( \vdash_\phi P \) (4)
  - 3, inversion on Rule (L:\PAR) \( \vdash_\phi G_1 + G_2 \) (5)
  - 5, inversion on Rule (L:GUARD) \( \vdash_\Lambda G_1 + G_2 \) (6)
  - 6, inversion on Rule (L:SUM) \( \vdash_\Lambda G_1 \) (7)
  - 7, formation on Rule (L:GUARD) \( \vdash_\phi G_1 \) (8)
  - 4, 8, formation on Rule (L:\PAR) \( \vdash_\phi P \parallel G_1 \) (9)
  - 2,9, IH \( \vdash_\phi Q \) (10)

**Case Rule (C:EXT):**

- **Assumption** \( \exists y. P \xrightarrow{(y)x} Q \) (1)
  - **Assumption** \( \vdash_\phi \exists y. P \) (2)
  - 1, inversion on Rule (C:EXT) \( P \xrightarrow{(x)} Q \) (3)
2, inversion on Rule (L:LOCAL) \( \vdash_o P \) (4)
3,4, IH \( \vdash_o Q \) (5)

Case Rule (C:Cong):

Assumption \( P_1 \xrightarrow{\alpha} P_2 \) (1)
Assumption \( \vdash_o P_1 \) (2)
1, inversion on Rule (C:Cong) \( P_1 \equiv_1 P_1' \) (3)
1, inversion on Rule (C:Cong) \( P_1' \xrightarrow{\alpha} P_2' \) (4)
1, inversion on Rule (C:Cong) \( P_2' \equiv_1 P_2 \) (5)
2, 3, Lemma [6.1] \( \vdash_o P_1' \) (6)
4, 6, IH \( \vdash_o P_2' \) (7)
5, 7, Lemma [6.1] \( \vdash_o P_2 \) (8)

Case Rule (C:Res):

Assumption \( \exists y. P \xrightarrow{\alpha} \exists y. Q \) (1)
1, inversion on Rule (C:Res) \( P \xrightarrow{\alpha} Q \) (2)
1, inversion on Rule (C:Res) \( y \notin \text{fv}(\alpha) \) (3)
Assumption \( \vdash_o \exists y. P \) (4)
4, inversion on Rule (L:LOCAL) \( \vdash_o P \) (5)
2, 5, IH \( \vdash_o Q \) (6)
6, formation on Rule (L:LOCAL) \( \vdash_o \exists y. Q \) (7)

D Appendix to Section 7

D.1 Preliminaries

Here we introduce a generalization of the notions of compositionality (cf. Def [4.2] and provide a definition of weak encoding.

Definition D.1 (Generalized Compositionality). Let \( \langle \models, \varphi_0 \rangle \) be a translation between two calculi \( \mathcal{L}_s = \langle P_s, \rightarrow_s, \approx_s \rangle \) and \( \mathcal{L}_t = \langle P_t, \rightarrow_t, \approx_t \rangle \). We say that \( \langle \models, \varphi_0 \rangle \) is compositional if and only if for every \( k \)-ary operator op of \( \mathcal{L}_s \) and for every subset of names \( N \), there exists a \( k \)-ary evaluation context \( C_{op}[\ldots] \) such that for each source process \( S_1, \ldots, S_n, n \geq 1 \), with \( \text{fn}(S_1, \ldots, S_n) = N \), it holds that
\[
\models \text{op}(S_1, \ldots, S_2) = C_{op}[\ldots] \models [S_1, \ldots, S_n]
\]

We will say that a weak encoding satisfies a more flexible notion of compositionality:

Definition D.2 (Weak encoding). Let \( \mathcal{L}_s = \langle P_s, \rightarrow_s, \approx_s \rangle \) and \( \mathcal{L}_t = \langle P_t, \rightarrow_t, \approx_t \rangle \) be calculi in the sense of Def [4.1]. A translation \( \langle \models, \varphi_0 \rangle \) of \( \mathcal{L}_s \) into \( \mathcal{L}_t \) is a weak encoding if it satisfies the following criteria:
1. **Name invariance:** For all $S \in P_s$ and substitution $\sigma$, we have $\lbrack S\sigma \rbrack = \lbrack S \rbrack \sigma'$, with $\varphi_{\Sigma_1}(\sigma(x)) = \sigma'(\varphi_{\Sigma_1}(x))$, for any $x \in V_s$.

2. **Weak compositionality:** For all $S \in P_s$, we have that there exist a context $C$ and a compositional translation $\langle \cdot \rangle$ (cf. Def. D.1) such that $\lbrack S \rbrack = C_{\text{fin}}(S)(\lbrack S \rbrack)$.

3. **Operational correspondence,** i.e., it is sound and complete:
   
   (a) **Soundness:** For all $S \in P_s$, if $S \rightarrow_s S'$, there exist $T \in P_t$ such that $\lbrack S \rbrack \Rightarrow_t T$ and $T \approx_t \lbrack S' \rbrack$.
   
   (b) **Completeness:** For all $S \in P_s$ and $T \in P_t$, if $\lbrack S \rbrack \Rightarrow_t T$, there exist $S'$ such that $S \rightarrow_s S'$ and $T \approx_t \lbrack S' \rbrack$.

### D.2 Correctness for $[\cdot]^+$

**Theorem D.3 (Name invariance for $[\cdot]^+$).** Let $P, \sigma$, and $x$ be an $s\pi^+$ process, a substitution satisfying the renaming policy for $[\cdot]^+$ (Def. 4.1b)), and a variable in $s\pi^+$, respectively. Then $\lbrack P\sigma \rbrack^+ = \lbrack P \rbrack^+ \sigma$ and $\varphi_{\Sigma_1^+}(\sigma(x)) = \sigma'(\varphi(x))$ for some $\sigma'$.

**Proof.** By structural induction on $P$. The only interesting case is for session initiation processes, and the proof is straightforward by considering that for service names $\varphi_{\Sigma_1^+}(a) = (a_1, a_2)$ (by Def. 7.2).

**Theorem D.4 (Compositionality of $[\cdot]^+$).** For every operator $\text{op}$ in $s\pi^+$ and for every $s\pi^+$ process $P_1, \ldots, P_n$, $n \geq 1$, such that $\text{fn}(P_1, \ldots, P_n) = N$, there exists a context $C_{\text{op}}^N$ in $s\pi$ such that

$$\lbrack \text{op}(P_1, \ldots, P_n) \rbrack^+ = C_{\text{op}}^N([P_1]^+, \ldots, [P_n]^+)$$

**Proof.** The proof proceeds by structural induction on the process $\text{op}(P_1, \ldots, P_n)$. All the cases that do not involve session initiation are trivial, since the mappings are homomorphic. The proof for session initiation is straightforward, since the new variables introduced are bound, therefore, the set of free variables of the process does not change.

Now, while translation $\langle [\cdot]^+, \varphi_{\Sigma_1^+} \rangle$ satisfies the previous two properties, this is not true for the operational correspondence property. In particular, if we consider two interacting process that are requesting and accepting a service $a$:

$$P = [\pi^m(z).P]^n \parallel [a^m_0(x).Q]^m$$

it easily noticeable that the translation $[P]^+$ does not provide the necessary restriction context $(\nu a_1 a_2)(\cdot)$ necessary in $s\pi$ to enable a reduction. Thus, $[\cdot]^+$ does not satisfy operational soundness.

To tackle this issue, we introduce the auxiliary mapping $\langle [\cdot]^+ \rangle$ (cf. Def. 7.2), which provides the necessary context for processes like $P$ to reduce. Thus, assume the translation $\langle [\cdot]^+, \varphi_{\Sigma_1^+} \rangle$ (notice that the renaming policy is kept unchanged). We will now prove that $\langle [\cdot]^+, \varphi_{\Sigma_1^+} \rangle$ is a weak encoding:

**Theorem D.5 (Name invariance for $[\cdot]^+$).** Let $P, \sigma$, and $x$ be an $s\pi^+$ process, a substitution satisfying the renaming policy for $[\cdot]^+$ (Def. 4.1b)), and a variable in $s\pi^+$, respectively. Then $\lbrack P\sigma \rbrack^+ = \lbrack P \rbrack^+ \sigma$ and $\varphi_{\Sigma_1^+}(\sigma(x)) = \sigma'(\varphi(x))$ for some $\sigma'$.

**Proof.** This theorem is a corollary of Thm. D.3. It follows straightforwardly by considering that $[\cdot]^+$ satisfies name invariance.

Now we prove that $\langle [\cdot]^+, \varphi_{\Sigma_1^+} \rangle$ is weakly compositional, in the sense of Def. D.2.2):
Theorem D.6 (Weak compositionality of $\llbracket \cdot \rrbracket^\ast$). Let $P$ and $C$ be an $\pi^\ast$ process and an $\pi^\ast$ context, respectively. Then $\llbracket P \rrbracket^\ast = C^{\text{sn}(P)}(\llbracket P \rrbracket^\ast)$, where $\llbracket \cdot \rrbracket^\ast$ is weakly compositional.

Proof. This proof is straightforward as a corollary of Thm. D.4 using Def. 7.2.

Theorem 7.3 (Operational correspondence of $\llbracket \cdot \rrbracket^\ast$). Statement on Page 39

Proof. We prove each item:

1. **Soundness**: The proof proceeds by induction on the length of the reduction with a case analysis on the last applied rule. The cases associated to the rules in Fig. 3 are straightforward, with $k = 1$. For the case associated to Rule $\text{Est}$ (cf. 4) the proof proceeds as follows:

$$P \longrightarrow^{\pi} Q \text{ with Rule } [\text{Est}] \quad (\text{Assumption}) \quad (1)$$

$$\llbracket P \rrbracket^\ast = (\overline{\pi^m(z).P'}_{l_j} \mid [a^m(x).P'' ]_m) \quad (\text{Assumption, with } a \in S_{\pi}) \quad (2)$$

$$Q = (\nu xy)(P' \langle y/z \mid P'' \rangle) \quad (\text{2, Rule } [\text{Est}]) \quad (3)$$

Suppose, without loss of generality, that $\text{fsn}(P) = \{a, b^1, \ldots, b^n\}$. By Def. 7.2 we obtain:

$$\llbracket (\overline{\pi^m(z).P'}_{l_j} \mid [a^m(x).P'' ]_m) \rrbracket^\ast = (\nu a_1 a_2)(\nu b_1 ^{i} b_2 ^{i}(a_1 \triangleleft m.a_2 \triangleleft l_j.a_1(z).\llbracket P'' \rrbracket^\ast \mid a_2 \triangleright \{m : a_2 \triangleright \{l_i : (nu xy)(\overline{\pi z}_2 y \mid \llbracket P'' \rrbracket^\ast \}}_{l_i \in \rho}\}$$

Then we have:

$$\rightarrow^{\pi} (\nu a_1 a_2)(\nu b_1 ^{i} b_2 ^{i}(a_1 \triangleleft l_j.a_1(z).\llbracket P'' \rrbracket^\ast \mid a_2 \triangleright \{l_i : (nu xy)(\overline{\pi z}_2 y \mid \llbracket P'' \rrbracket^\ast \}}_{l_i \in \rho}\})$$

$$\equiv^{\pi} (\nu a_1 a_2)(\nu b_1 ^{i} b_2 ^{i}(nu xy)(a_1(z).\llbracket P'' \rrbracket^\ast \mid (\overline{\pi z}_2 y \mid \llbracket P'' \rrbracket^\ast )}, \text{ note that } x, y \notin \text{fsn}(a_1(z).\llbracket P'' \rrbracket^\ast )$$

$$\rightarrow^{\pi} (\nu a_1 a_2)(\nu b_1 ^{i} b_2 ^{i}(nu xy)(\llbracket P'' \rrbracket^\ast \langle y/z \mid P'' \rangle = (\nu x y)(\llbracket P'' \rangle^\ast$$

thus yielding the expression we required to complete the proof; notice that $k = 3$.

2. **Completeness**: We proceed by induction on the reduction with a case analysis in the last applied rule. Because session establishment is encoded by additional branching/selection operators, the only interesting case is that of Rule $\text{Sel}$ (cf. 4, Page 38):

$$\llbracket P \rrbracket^\ast \longrightarrow^{\pi} R \text{ using Rule } [\text{Sel}] \quad (\text{Assumption}) \quad (1)$$

$$\llbracket P \rrbracket^\ast \equiv^{\pi} (\nu u_1 u_2)(\nu w_1 ^{i} w_2 ^{i}(u_1 \triangleleft l_j.P' \mid u_2 \triangleright \{l_i : P'' \}}_{i \geq 0}), i \geq 0 \quad (\text{Assumption}) \quad (2)$$

Considering Def. 7.2 we can distinguish two cases:

- **Case** $u \in \text{fsn}(P)$: In this case we assume that $P = (\overline{\pi^m(z).P'}_{l_j} \mid [w^m(x).Q]'_m$ and proceed with a reduction as in the presented in the soundness case.

- **Case** $u \notin \text{fsn}(P)$: This case is straightforward as the translation is homomorphi.
D.3 Correctness for $\llbracket \cdot \rrbracket_f$

Theorem 7.8 (Name Invariance for $\llbracket \cdot \rrbracket_f$). Statement on Page 42.

Proof. The proof for this theorem proceeds by induction on the structure of $P$ as follows. We only show the cases for $P = 0$ and $P = \overline{x\cdot v.Q}$; all the other cases are straightforward:

Case $P = 0$:

(i) By Fig. 15 we have that $\llbracket 0 \rrbracket_{\pi} = \top$.

(ii) Since there are no variables in $\top$ then $\top\sigma = \top$.

(iii) By (i),(ii) we have that $\llbracket 0 \rrbracket_{\pi\sigma} = \llbracket 0 \rrbracket_{\pi\sigma}$.

Case $P = \overline{x\cdot v.Q}$:

(i) By Fig. 15 and Def. 7.7(b) we have that:
$$\llbracket \overline{x\cdot v.Q} \rrbracket_{\pi\sigma} = \text{snd}(x; v)\sigma \parallel \forall \epsilon (\text{ch}(x; \epsilon)\sigma \mid \{ x: f_x \}\sigma \otimes \text{rcv}(f_x, v; \epsilon)\sigma \rightarrow \llbracket Q\rrbracket_f^\pi)$$

(ii) By Fig. 15 and Def. 7.7(b) we have that:
$$\llbracket \overline{x\cdot v.Q} \rrbracket_{\pi\sigma} = \text{snd}(x; v)\sigma \parallel \forall \epsilon (\text{ch}(x; \epsilon)\sigma \mid \{ x: f_x \}\sigma \otimes \text{rcv}(f_x, v; \epsilon)\sigma \rightarrow \llbracket Q\rrbracket_f^\pi)$$

(iii) By IH, we have that $\llbracket Q \rrbracket_f^\pi \sigma = \llbracket Q\rrbracket_f^\pi$.

(iv) By (i),(ii),(iii) we have that $\llbracket \overline{x\cdot v.Q} \rrbracket_f^\pi \sigma = \llbracket \overline{x\cdot v.Q} \rrbracket_f^\pi$.

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

Theorem 7.9 (Compositionality of $\llbracket \cdot \rrbracket_f$). Statement on Page 42.

Proof. The proof proceed by induction on the structure of $P$ and a case analysis on the grammar in Def. 4.7. We only show the cases for $P = 0$ and $P = \overline{x\cdot v.Q}$; all the other cases are straightforward:

Case $P = 0$:

- **Subcase** $E[\cdot] = R | ::$
  
  (i) By Fig. 15 $\llbracket E[0] \rrbracket_f^\pi = \llbracket R \rrbracket_f^\pi | \llbracket 0 \rrbracket_f^\pi$.
  
  (ii) By (i) and Fig. 15 we have that $\llbracket R \rrbracket_f^\pi | \llbracket 0 \rrbracket_f^\pi = \llbracket E[0] \rrbracket_f^\pi$.

- **Subcase** $E[\cdot] = \cdot | R$: Analogous to the previous one.

- **Subcase** $E[\cdot] = (\nu xy)(\cdot)$: By Fig. 15 and structural congruence see that $(\nu xy)(0) \equiv_\pi 0$, thus $\llbracket E[0] \rrbracket_f^\pi | \llbracket 0 \rrbracket_f^\pi = \llbracket E[0] \rrbracket_f^\pi$.

Case $P = \overline{x\cdot v.Q}$:

- **Subcase** $E[\cdot] = R | ::$
  
  (i) By Fig. 15 $\llbracket E[\overline{x\cdot v.Q}] \rrbracket_f^\pi = \llbracket R \rrbracket_f^\pi | \llbracket x\cdot v.Q \rrbracket_f^\pi$. 

78
(ii) By (i) and Fig. 15, we have that \( \| R \|_f \| \| P \|_v \|_v = \| E \|_f \| \| \| P \|_v \|_v \|_v = \| E \|_f \| \| P \|_v \|_v \|_v = \| E \|_f \| \| P \|_v \|_v \|_v \).

- **Subcase** \( E[\cdot] = \cdot \mid R \): Analogous to the previous one.

- **Subcase** \( E[\cdot] = (\nu w)(\cdot) \):
  (i) By Fig. 15 \( \| E \|_f \| \| P \|_v \|_v = \| \| P \|_v \|_v \|_v \).
  (ii) By (i) and Fig. 15 we have that \( \exists w, z. \| \| P \|_v \|_v = \| \| \| P \|_v \|_v \|_v = \| \| P \|_v \|_v \|_v \).

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

\[ \square \]

**Theorem 7.12 (Operational Correspondence for \( \| \|_f \)). Statement on Page 44**

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

1. **Soundness:**

   - The proof maintains the structure of the previous proof of operational correspondence (cf. Thm. 4.12). The only new case is Rule [Est]:
     (i) Assume then that:
     \[
     ([\alpha^2(x).P]_{P_E} \mid [\alpha^2(x).Q]_{P_E}) \rightarrow (\nu x y)(P\{y/u\} \mid Q) \quad (l_1\in \rho)
     \]
     (ii) Let \( S = \| ([\alpha^2(x).P]_{P_E} \mid [\alpha^2(x).Q]_{P_E}) \|_f \), then by Def. 7.7 we have:
     \[
     S = \exists m. (\text{out}(\varepsilon; \{\langle m, l_1 \rangle\})_{P_E}) \parallel \forall u, v (\text{out}(\varepsilon; \{\langle m, v, u, l_2 \rangle\})_{P_E}) \rightarrow
     \text{out}(\varepsilon; \{\langle v, u \rangle\}_{P_E}) \parallel \text{out}(\varepsilon; \{\langle v, u \rangle\}_{P_E}) \parallel \| P_{f_\cup(u:v:u)} \|
     \]
     \[
     \equiv \exists x, y, m. (\text{out}(\varepsilon; \{\langle x, y \rangle\})_{P_E}) \rightarrow \exists z, n \ (\text{out}(\varepsilon; \{\langle z, n \rangle\})_{P_E}) \rightarrow \| \text{loc}(\varepsilon; \{\langle z, x, y, l_2 \rangle\}) \| \rightarrow
     \text{out}(\varepsilon; \{\langle x, y, l_2 \rangle\}) \rightarrow \| P_{f_\cup(x:y)} \|
     \]
     \[
     \equiv \exists x, y, m. (\text{out}(\varepsilon; \{\langle x, y \rangle\})_{P_E}) \rightarrow \exists z, n \ (\text{out}(\varepsilon; \{\langle z, n \rangle\})_{P_E}) \rightarrow \| P_{f_\cup(x:y)} \|
     \]
     (iii) Since \( m \) is a fresh variable that does not exist in \( P \) or \( Q \) and \( v \) is not free in \( P \) (by Def. 7.7), we can conclude:
     \[
     \exists x, y, m. (\| P\{x, y/v, u\}\|_{f_\cup(x:y)} \| \| \{x:y\}\|_{f_\cup(x:y)} \| \| Q\|_{f_\cup(x:y)} )
     \equiv \exists x, y, m. (\| P\{x, y/v, u\}\|_{f_\cup(x:y)} \| \| \{x:y\}\|_{f_\cup(x:y)} \| \| Q\|_{f_\cup(x:y)} )
     \]
     \[
     = (\| \nu(xy)(P\{y/u\} \mid Q)\|_{f_\cup(x:y)})
     \]

79
• 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 $P = (\nu xy)(x.v.P' | y(z).P'')$.

(ii) By (i) $P \rightarrow_\pi (\nu xy)(P' | P''\{v/z\})$ using Rule $\text{[Com]}$.

(iii) By definition of $[.]_f^g$:

$$[P]_f^g = \exists x, y. (\{x:y\} \land \text{lcc}(x; v) \land \forall e(ch(x; e) \land \{x:f_x\} \land \text{snd}(f_x; z) \rightarrow [P']_f^g) \land \forall z(ch(y; e) \land \{y:f_x\} \land \text{snd}(f_x; z) \rightarrow [P'']_f^g))$$

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

$$[P]_f^g \equiv_\iota = \exists x, y. (\{x:y\} \land \text{lcc}(x; v) \land \forall e(ch(x; e) \land \{x:f_x\} \land \text{snd}(f_x; z) \rightarrow [P']_f^g) \land \forall z(ch(y; e) \land \{y:f_x\} \land \text{snd}(f_x; z) \rightarrow [P'']_f^g))$$

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

$$\equiv_\iota \exists x, y. (\{x:y\} \land \{P'\}_f^g \land \{P''\}_f^g)$$

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.

Let $S = \{([\pi^2(u).P]_{l_1}) | [a_0^p(x).Q]_{l_2})\}_f^g$:

$$S = \exists m. (\text{out}(e; \{\langle m, l_1 \rangle \}_{p(l_2)}) \land \forall u, v (\text{out}(e; x(l_1)) \land \text{out}(e; \{\langle m, v, u, l_2 \rangle \}_{p(l_2)}) 
\rightarrow 
\text{out}(e; \{\langle v, u \rangle \}_{p(l_2)}) \land \{x:v\}_f^g \land \{P'\}_f^g \land \{P''\}_f^g)$$

$$\land \exists x, y. (\forall z, n (\text{out}(e; x(l_2)) \land \text{out}(e; \{\langle z, n \rangle \}_{p(l_2)} \land \text{loc}(n; e) \rightarrow \text{out}(e; \{\langle z, x, y, l_2 \rangle \}_{p(n)}) \land \forall v (\text{out}(e; x(l_2)) \land \text{out}(e; \{\langle x, y \rangle \}_{p(l_2)}) 
\rightarrow \{Q\}_f^g \land \{x:y\}_f^g))$$

By building a similar reduction for the case of soundness, it is possible to see that in one reduction we will have process $S \in \{[P]_f^g\}$, and that in two reductions and one structural congruence step we obtain $[Q]_f^g$, as desired.

\[
\]

80
Case \( P = [\rho^i_n(x), Q]^m \): The derivation tree is given in Fig. 16 (Page 45).
Case \( P = [\pi v.Q]^m \): The derivation tree is given in Fig. 19.
Case \( P = \pi v.Q \): The derivation tree is given in Fig. 20.
Case \( P = x(y).Q \): The derivation tree is given in Fig. 21.
Case \( P = x \triangleq l, P \): The derivation tree is given in Fig. 22.
Case \( P = x \triangleright \{ l_i : Q_i \}_{i \in I} \): The derivation tree is given in Fig. 23. Note that when using the inductive hypothesis, we previously need to use \( n \) steps with (L.Par).
Case \( P = v^? (Q) : (R) \): The proof is trivial as all the variables of an equality are unrestricted; we omit the derivation tree.

\[ \square \]
Figure 21: Typing derivation for \([x(y).Q]_f^S\), [1] = (L:COMB), (L:COMB), (L:Pred), (L:Pred), (L:Pred).

Figure 22: Typing derivation for \([x \lor l.Q]_f^S\), [1] = (L:COMB), (L:COMB), (L:Pred), (L:Pred), (L:Pred).

Figure 23: Typing derivation for \([x \lor \{l_i : Q_i\}_{i \in I}]_f^S\), [1] = (L:COMB), (L:COMB), (L:Pred), (L:Pred), (L:Pred).

82