Parameterized, Concurrent Session Types for Asynchronous Multi-Actor Interactions

Minas Charalambides[a], Peter Dinges[b], Gul Agha[c]

[a] *charala1@illinois.edu*
[b] *pdinges@acm.org*
[c] *agha@illinois.edu*

Abstract

Session types have been proposed as a means of statically verifying implementations of communication protocols. Although prior work has been successful for some classes of protocols, it does not cope well with parameterized, multi-actor scenarios with inherent asynchrony. For example, the sliding window protocol is not expressible in previously proposed session type notations. This article defines System-A: a novel session type system, as well the associated programming language that together overcome many of the limitations of prior work. With explicit support for asynchrony and concurrency, as well as multiple forms of parameterization, we demonstrate that System-A can be used for the static verification of a large class of asynchronous communication protocols.

Keywords: session types, type theory, distributed, static, parameterized, multi-party, asynchrony, concurrency, parallel, actors, communication

1. Introduction

Session types [35] are a means of expressing the order and type of messages exchanged by concurrently executing processes. In particular, session types can be used to statically check if a group of processes communicates according to a given specification. In these systems, a *global type* specifies the permissible sequences of messages that participants may exchange in a given *session*, as well as the types of these messages. The typing requires the programmer to provide the *global type*. A *projection* algorithm then generates the restrictions implied by the global type for each participant. Such restrictions are called *end-point types* or *local types* and describe the

expected behavior of the individual participants in the protocol. The actual program code implementing the behavior of a participant is checked for conformance against this localized behavior specification. We are interested in generalizing prior work on session types to typing coordination constraints in *parameterized* actor [1] programs, which can then be enforced, e.g., with *Synchronizers* [27, 28, 24] or other ways [42, 2].

Typing coordination constraints in actors requires addressing two problems. First, asynchronous communication leads to delays that require considering arbitrary shuffles. Second, we need to consider *parameterized* protocols. For example, assume two actors communicating through a sliding window protocol [51]: the actors agree on the length of the window (i.e., the number of messages that may be buffered) and then proceed to a concurrent exchange of messages. Prior work on session types is not suitable for typing interactions such as the sliding window protocol. The reason for this limitation is that their respective type languages depend on other formalisms for type checking (such as typed λ-calculus [3] or System T [32]), and these formalisms do not support a concurrency construct.

Contributions. We present a programming language along with a session type system that overcomes many of the aforementioned limitations through the use of parameters and novel constructs. The session types were originally introduced in our 2012 FOCLASA paper [17], which serves as the foundation of the present article. The three primary extensions are (a) the introduction and formal definition of Lang-A, a programming language for expressing protocols typeable in our session type system, System-A; (b) an inference algorithm that derives local System-A types from Lang-A programs, and (c) the formal treatment of the type system.

Overall, our work on System-A makes the following contributions: (i) the introduction of *parameterized* constructs for expressing asynchrony, concurrency, sequence, choice and atomicity in protocols; (ii) a projection mechanism that extracts local type constraints on individual actors from the global type; (iii) a formalization of the conditions under which conformance of these constraints to the global type is assured; (iv) a normalization algorithm for local types, which allows equivalence testing; and (v) trace semantics of System-A and Lang-A. We furthermore supply the formal proofs of various useful properties, such as those concerning items (iii) and (iv), as well as standard sanity checks of the type system.

Limitations. We do not address dynamic actor creation in this article, and briefly discuss the related difficulties in Section 11. We furthermore omit support for session delegation, and do not deal with issues of progress. Finally, we do not consider overlapping indexed names when nested in multiple operators. This disallows some cases, such as all-to-all communication. A more accurate description of how we restrict the use of indices is given in Section 7.

2. Related Work

Session types [35, 54, 50, 34] originate from the context of π-calculi as statically derivable descriptions of process interaction behaviors. In two-party sessions, they allow us to statically verify that the participants have compatible behavior by requiring *dual* session types, that is, behaviors where each participant expects precisely the message sequence that the other participant sends and vice versa. Extensions to session types support asynchronous message passing [36] and introduce subtyping [29] for a looser notion of type compatibility. Session types have been integrated into functional [52, 47] and object-oriented [23, 38, 31] languages among others. Other extensions deal with evolving system specifications using transformations [25]. Exception handling, which allows the participants of a protocol to escape the normal flow of control and coordinate on another, has also been considered [14, 12]. The present article introduces a novel combination of two enhancements to session types: we parameterize both the number of participants, and the type constructs, including those introducing asynchrony. This greatly extends the applicability of types.

Asynchronous Multi-Party Sessions. Many real-world protocols involve more than two participants, which makes their description in terms of multiple two-party sessions unnatural. To overcome this limitation, Honda et al. [36] extend session types to support multiple participants: A *global type* specifies the interactions between all participants from a global perspective. A projection algorithm then mechanically derives the behavior specification of each individual participant, that is, its *local type*.

The notion of a global specification and the associated correctness requirements for projection were first studied by Carbone et al. [11], although in the context of concrete implementations rather than session types; Bonelli's work on multi-point session types [9] treats multi-party protocols from the local

perspective only. Bettini et al. [7] allow multi-party sessions to interleave and derive a type system guaranteeing global progress. Gay and Vasconcelos [30] consider subtyping in presence of asynchrony. Recent work by Carbone and Montesi [13] takes a novel approach to asynchrony by treating programs from the global perspective and regaining concurrent composition through a suitable swap relation. Although it obviates the need for an explicit concurrent operator, their approach does not support parameterized programs. In contrast, we include a parameterized concurrent construct in our types. Concurrent composition is also treated in the work of Kouzapas et al. [39], who apply session type discipline to π-calculus. An event-driven approach to asynchrony is explored by Hu et al. [37]. However, neither the system of Kouzapas, nor that of Hu handles parameterized asynchrony.

The type systems introduced by Puntigam [49, 48] deal with asynchrony in the context of actors. His approach has the benefit of not utilizing global types; instead, given an actor's specification, the system ensures its proper use. To the best of our knowledge, this is also the only relevant body of work where the language supports dynamic actor creation, which is something that System-A does not handle. However, it is important to note that Puntigam's systems ensure safety properties on a per-actor basis, which is (naturally, due to the absence of global types) somehow limited in scope. Parameters are also not considered.

The present article builds on the foundation of a global protocol specification and its projection onto local behaviors, as in the work of Honda et al. [36]. Unlike Honda's approach (but following Castagna et al. [15, 16]), we simplify the notation for global types by replacing recursion with the Kleene star and limiting each pair of participants to use a single bidirectional channel. We introduce an explicit shuffle operator to preserve the commutativity of message arrivals that can be achieved using multiple channels. Explicit shuffles also reduce the need for a special subtyping relationship that allows the permutation of (Lamport-style) concurrent asynchronous events for optimization [43].

Following Castagna's global type syntax further, we support join operations. Joins cannot be expressed in Honda et al.'s global types because of the linearity requirement. However, as Deniélou and Yoshida remark [22], join operations can only describe series–parallel graphs. Protocols such as the alternating bit protocol [41] that require interleaved synchronization between two processes consequently cannot be expressed in our global type language. Our choice to not support generic graph structures as global types

is founded on the desire to support parameterization and, at the same time, keep the language understandable; it remains unclear to us how to visualize parameterized graphs in an intuitive fashion.

Parameterized Session Types. Our major extension over Honda et al. and Castagna et al.'s work is the introduction of parameters. The main inspiration for our parameterization of session types is the work of Yoshida et al. [53] and Bejleri [5, 6]. Yoshida et al. augment the global types of Bettini et al. [7] with primitive recursive combinators to obtain dependent types that support the parameterization of the repetition count and the connection topology. This allows, for example, using a single global type for a highly participant-count dependent butterfly network. Static verifiability—without instantiating the type parameters—is maintained by projecting onto parameterized local types that allow syntactic comparisons. Deniélou and Yoshida [21] achieve parameterization by means of quantification over behavior specifications they call *roles*. Like Bettini et al., and unlike System-A, neither Deniélou et al. nor Yoshida et al. support arbitrary, concurrency-induced shuffles in their global and local types. While Bettini et al. regain concurrent composition through the interleaving of global types, it is unclear how the results transfer to the other two approaches.

Realizability of Sessions. We draw our ideas for global type realizability criteria primarily from the approach of Castagna et al. [16]. However, the presence of parameters in our operators makes realizability harder to tackle in a purely structural manner; we thus apply our criteria to traces instead. Some of our ideas for realizability parallel the work of Lanese et al. [40], although our criteria are stronger in some cases, as for example, our sequencing criterion handles repetitions. The notion of a *distinctive point* for implementing choice (Section 9) is also employed in their work, where it is called *unique point of choice*, but in a simpler context (absence of parameters). Basu et al. [4] derive precise realizability criteria for choreographies. Although not concerned with parameterization, their work demonstrates conditions that are both necessary and sufficient. In contrast, our realizability results are conservative, that is, we only state sufficient conditions.

Modeling of Multi-Party Protocols. Formalisms for describing multi-party communication protocols have been studied in the context of designing distributed systems and cryptography protocols. As modeling tools, the formalisms provide ways to check a protocol for desired properties [55], or to

synthesize such protocols [46]. In contrast to session types, those formalisms lack ways to statically verify the compliance of an actual protocol implementation against the specification. Deniélou and Yoshida [22] discuss session types and their relation to work on distributed systems and cryptography protocols in greater depth.

3. Motivation

This section motivates our approach with a discussion of the sliding window protocol, a case of locking–unlocking, and some cases of limited resource sharing. We demonstrate how the behavior of such protocols can be described in System-A.

The Sliding Window Protocol. Assume that an actor a sends messages of type m to an actor b, which acknowledges every received message with an *ack* message. The protocol determines that at most n messages can be unacknowledged at any given time, so that a ceases sending until it receives another *ack* message. In this example, the window size n is a parameter, which means that we need a way to express the fact that n sending–acknowledging events can be in transit at any given instant in time. The global type of the protocol is as follows:

$$\underbrace{\left(a \xrightarrow{m} b \,;\, b \xrightarrow{ack} a\right)^* \parallel \left(a \xrightarrow{m} b \,;\, b \xrightarrow{ack} a\right)^* \parallel \cdots \parallel \left(a \xrightarrow{m} b \,;\, b \xrightarrow{ack} a\right)^*}_{n \text{ times}}$$

where $a \xrightarrow{m} b$ denotes that a sends a message of type m to b. The operator ; is used for sequencing interactions. Operator $\parallel$ is used for the concurrent composition of its left and right arguments, while the Kleene star has the usual semantics of an unbounded—yet finite—number of repetitions.

The above type can be expressed using the notation of Castagna et al. [15, 16], albeit with a fixed window size n. In System-A on the other hand, we can parameterize the type in n and statically verify that participants follow the protocol without knowing its runtime value. Using $\parallel_{i=1}^{n}$ to denote the concurrent composition of n processes, we obtain the following type in our notation:

$$\prod_{i=1}^{n} \left(a \xrightarrow{m} b \,;\, b \xrightarrow{ack} a\right)^*. \tag{1}$$

Locking / Unlocking. Consider a set of n client actors $c_{1..n}$, each of which needs to acquire exclusive access to a server s, by sending it a *lock* message. The server replies with *ack*, the client uses its services (not shown) and then unlocks it by sending an *unlock* message, at which point the next client can do the same. Using $\otimes$ to denote an arbitrary, atomic reordering of terms, the following type describes the locking–unlocking protocol for a fixed number of participants:

$$(c_1 \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_1 \;;\; c_1 \xrightarrow{unlock} s) \;\otimes\; \cdots \;\otimes\; (c_n \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_n \;;\; c_n \xrightarrow{unlock} s).$$

This formula expresses that any ordering of the $(c_i \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_i \;;\; c_i \xrightarrow{unlock} s)$ sequences is acceptable. To support a dynamic network topology, the number of participants should be a parameter. The following is the locking–unlocking example in System-A, where conformance to the protocol is statically verifiable without knowledge of the runtime value of n:

$$\bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_i \;;\; c_i \xrightarrow{unlock} s). \tag{2}$$

Limited Resource Sharing. In this scenario, a server s grants two clients c_1 and c_2 exclusive access to a set of n resources. At any given point, a maximum of n resources can be locked, but the relevant lock–ack–unlock messages from both clients can be interleaved in any way. Following is the global type for this situation:

$$\Big\|_{i=1}^{n} \big(c_1 \xrightarrow{lock_i} s \;;\; s \xrightarrow{ack_i} c_1 \;;\; c_1 \xrightarrow{unlock_i} s \;\oplus\; c_2 \xrightarrow{lock_i} s \;;\; s \xrightarrow{ack_i} c_2 \;;\; c_2 \xrightarrow{unlock_i} s\big)^*.$$

The concurrent composition is parameterized in n, the number of resources. Each sequence of lock–ack–unlock messages is also parameterized in i, which ranges from 1 to n and signifies the resource it refers to. This is necessary to ensure realizability of the protocol, because in the case of multiple outstanding requests, it allows the participants to disambiguate the responses they receive. Each concurrent instance subsumed by the $\|_{i=1}^{n}$ operator consists of a loop (Kleene star) which entails a choice, indicated by $\oplus$. Either c_1 gets access to a resource, or c_2, and this happens repeatedly.

These type operators may be combined to express more complicated resource sharing. For instance, consider another version of the previous ex-

ample, where not only the number n of resources, but also the number k of clients is a parameter:

$$\prod_{i=1}^{n} \Big(\bigoplus_{j=1}^{k} \big(c_j \xrightarrow{lock_i} s \;;\; s \xrightarrow{ack_i} c_j \;;\; c_j \xrightarrow{unlock_i} s \big) \Big)^{*}. \tag{3}$$

4. Global Types

A global type describes a protocol to which the whole system must adhere. The examples in Section 3 are all global types since they describe the behavior of all participants. This section formalizes the language of global types in System-A. We first provide the syntax and intuitive meaning of the operators. Then, in Section 4.2, we define their formal semantics.

4.1. Syntax of Global Types

Table 1 presents the grammar that generates the syntactic category $\mathscr{G}$ of global types. The elements of $\mathscr{G}$, instances of global types, will be denoted by variations of the variable G. Intuitively, the rules capture the following concepts:

(G-Interaction) denotes the sending and receiving of a message. For instance, $p_1 \xrightarrow{t} p_2$ means that participant p_1 sends a message of type t to participant p_2.

(G-Seq) is used for the sequential composition of events.

(G-Choice) denotes exclusive choice between the arguments. For $G_{1,2} \in \mathscr{G}$, $G_1 \oplus G_2$ means that only one of G_1, G_2 will be executed.

(G-Paral) means that the arguments run concurrently. Interleavings are allowed, as long as the order established by the ; operator is respected. For example, $(a \xrightarrow{t_1} b \;;\; a \xrightarrow{t_2} c) \parallel c \xrightarrow{t_3} b$ means that all interleavings ABC, ACB, CAB are possible, where A $= (a \xrightarrow{t_1} b)$, B $= (a \xrightarrow{t_2} c)$ and C $= (c \xrightarrow{t_3} b)$. Note that B is not allowed to precede A because of how ; orders them (hence BAC, BCA and CBA are not valid interleavings in this example).

(G-Shuffle) means that both arguments are executed atomically, in an unspecified order. Formally, $G_1 \otimes G_2 \equiv (G_1 \,;\, G_2) \oplus (G_2 \,;\, G_1)$ with the $\equiv$ relation denoting semantic equivalence (Table 2).

(G-KleeneStar) has the usual semantics of zero or more repetitions of the argument. We assume a finite number of repetitions.

Table 1: *The syntax of global types. The auxiliary symbols appearing in the grammar have the following domains:* $i \in \mathit{IndexNames}$; $n, n_1, n_2 \in \mathit{ParamNames} \cup \mathbb{N}$; $a, b \in \mathit{ActorNames} \cup \{\alpha_j \mid \alpha \in \mathit{ActorNames},\, j \in \mathit{IndexNames}\}$; *and* $m \in \mathit{MsgNames} \cup \{\mu_j \mid \mu \in \mathit{MsgNames},\, j \in \mathit{IndexNames}\}$.

$\mathscr{G} ::=$	$a \xrightarrow{m} b$	(G-Interaction)	$\mid$	$(\mathscr{G})$	(G-Paren)
$\mid$	$\mathscr{G} \,;\, \mathscr{G}$	(G-Seq)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\circledcirc}} \mathscr{G}_i$	(G-Seq-N)
$\mid$	$\mathscr{G} \oplus \mathscr{G}$	(G-Choice)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\bigoplus}} \mathscr{G}_i$	(G-Choice-N)
$\mid$	$\mathscr{G} \parallel \mathscr{G}$	(G-Paral)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\Vert}} \mathscr{G}_i$	(G-Paral-N)
$\mid$	$\mathscr{G} \otimes \mathscr{G}$	(G-Shuffle)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\bigotimes}} \mathscr{G}_i$	(G-Shuffle-N)
$\mid$	$\mathscr{G}^n$	(G-Exp)	$\mid$	$\mathscr{G}^*$	(G-KleeneStar)

The n-ary versions of the operators express behaviors where the value of n, n_1, and n_2 are unknown at compile time. Intuitively, the rules (G-Seq-N), (G-Choice-N), (G-Paral-N), and (G-Shuffle-N) apply the respective binary operator $n_2 - n_1$ times, generating a global type for each of the $n_2 - n_1 + 1$ values of i. (G-Exp) denotes the n-fold, sequential repetition of the argument. Note that for known parameter values, these expansions can take place during compilation.

All of the operators are commutative, with the exception of sequencing. All operators are furthermore associative, with the exception of shuffling. In particular,

$$\overset{n}{\underset{i=1}{\bigotimes}} G_i \neq \big(\ldots\big((G_1 \otimes G_2) \otimes G_3 \ldots\big) \otimes \ldots \otimes G_n\big)$$

because the meaning of $\bigotimes_{i=n_1}^{n_2} G_i$ is that all arguments G_i are executed atomically, but in an unspecified order. Instead, the right-hand side above prevents, for example, G_3 from occurring between G_1 and G_2.

The distinction between the Kleene star and exponentiation is fundamental. The use of G^n means that the protocol conformance checker will have to prove that the system is correct for any fixed value of the parameter n. G^* on the other hand means an unbounded number of repetitions of G. There is no parameter fixing this number, and it may be different from instance to instance of the Kleene star and/or among executions of the same program with the same run-time values for its parameters. The Kleene star entails a choice as to when to exit the loop, a difference that becomes clearer in Section 9.5.

4.2. Semantics of Global Types

This section formalizes the intuitions introduced earlier by defining the trace semantics of global types. The traces of a global type $G \in \mathscr{G}$ capture the permissible sequences of messages that participants may exchange. Later, in Section 9, we define properties of G's traces that ensure that a program with local types derived from G adheres to the protocol specified by G. We use the following definitions:

Definition 1 (Event)**.** An *event* is a single interaction $p_1 \xrightarrow{m} p_2$.

Definition 2 (Trace)**.** A *trace* is a finite sequence of events and is of the form $e_1 \,;\, e_2 \,;\, \ldots \,;\, e_k$.

In the operational semantics shown in Table 2, a configuration is a tuple of the form (T, X), where T denotes the trace produced so far, and X is a set of concurrently executing processes, which we need to model concurrent composition and other constructs. Each such process is of the form $\langle G \rangle$ where $G \in \mathscr{G}$ is a global type as defined in Table 1. To make the presentation cleaner, our rules omit T when it is clear that it remains the same. Notice that rules after (GS-Trace) do not mention T — (GS-Trace) is the only rule that produces an event. The rule makes a non-deterministic choice among processes whose prefix is an interaction and appends the respective event to T. In general, if more than one rewriting rule applies, we assume that the system makes a non-deterministic choice.

Definition 3 (Traces of a Global Type)**.** The set of traces $tr\,(G)$ producible by a global type $G \in \mathscr{G}$ consists of all traces that can be derived by applying the semantic rules in Table 2 to the initial configuration $\big(\epsilon, \{\langle G \rangle\}\big)$ until termination. That is, all different values T can have when G terminates, when T's initial value is set to ϵ. We say that G terminates when the set X in the semantic configuration contains nothing but $\big\langle \tau \big\rangle$ processes.

In the semantics, we assume the existence of a function $v(\cdot)$ that evaluates its argument expression without side-effects. We use v to retrieve the values of type parameters, constants, and simple expressions involving these. Additionally, notice rule (GS-Init), which appends τ to G — this enables termination as per Definition 3. In a slight abuse of notation, G can take the value τ for the purposes of this section.

Many of the rules produce special markers to help distinguish execution paths and coordinate concurrent processes. When the marker is first produced, it is given a fresh subscript k that identifies the particular rewriting step that caused the marker to appear. The superscript is a counter, useful in cases such as the join marker j_k^x. The latter is used in (GS-Join) to merge concurrent processes when all x of them reach the join with the same k.

Rules (GS-Star) and (GS-Exp) perform the standard rewritings of Kleene star and exponentiation, respectively, in terms of other operators. Rule (GS-Paral) produces two processes to model concurrent composition. These processes merge at a special join marker j_k^x, as described by rule (GS-Join). Rule (GS-Choice) also produces two processes, one for each branch of the choice operator. Both are preceded by a special selection marker, which is used to make a non-deterministic choice in (GS-Ch-Done).

Note that X is an unordered set, and hence in (GS-Ch-Done) the process to execute (denoted by G_1 in the rule) is chosen non-deterministically. Because (GS-Star) introduces choice, this has the important implication that the rules might execute Kleene star loops an infinite number of times. To overcome such non-termination issues, we make a fairness [26] assumption whereby rule (GS-Ch-Done) does not permanently "ignore" a branch that shows up repeatedly. This ensures traces of finite (yet unbounded) length.

Shuffling is more complicated because it entails the atomic execution of all the arguments of the operator. (GS-Shuffle) produces processes that enclose the arguments in pairs of *s-on* and *s-end* markers, followed by a join. The "on" marker allows a non-deterministic choice of which branch to begin executing, as per (GS-Shf-Begin). This switches the rest of the processes to

s-off, which is only reversed when the *s-end* marker is reached (GS-Shf-End). In that case, the completed branch stalls at a join marker j_k^x, while the remaining *s-on*'s have their counter reduced to $x - 1$.

The parameterized versions of the operators are resolved similarly. First, rule (GS-Op) introduces a sp_k^x marker that records the number of processes to spawn, and then the appropriate -N rule takes it from there. We use the notation $R[x/y]$ to denote the substitution of all free instances of y in R by x. For example, (GS-Paral-N) spawns one process for the current value of the index i, and another one with the same operator having the starting value incremented by one. The latter will cause repeated applications of the rule, which stop with (GS-Op-End).

Table 2: *The semantics of global types. The rules transform configurations (T, X), where T denotes the trace produced so far, and X is the set of concurrently executing processes $\langle G \rangle$, $G \in \mathscr{G}$. For brevity, the rules omit unchanged elements. Function $v(\cdot)$ evaluates expressions; $[\cdot/\cdot]$ denotes substitution of free variables; and s-on etc. are special markers that steer the execution. See Table 1 for the domains of the remaining symbols.*

(GS-Init)	$\epsilon,\ \{\langle G \rangle\} \ \rightsquigarrow\ \epsilon,\ \{\langle G; \tau \rangle\}$	
(GS-Trace)	$T,\ X \cup \langle a \xrightarrow{m} b\ ;\ G \rangle \ \rightsquigarrow\ T \cdot (a \xrightarrow{m} b),\ X \cup \langle G \rangle$	
(GS-Star)	$X \cup \langle G^*\ ;\ G' \rangle \ \rightsquigarrow\ X \cup \langle (G\ ;\ G^*\ ;\ G') \ \oplus\ G' \rangle$	
(GS-Exp)	$X \cup \langle G^n\ ;\ G' \rangle \ \rightsquigarrow\ X \cup \langle \big(\mathop{\circledcirc}_{i=1}^{n} G\big)\ ;\ G' \rangle$	
(GS-Paral)	$X \cup \langle (G_1 \parallel G_2)\ ;\ G' \rangle \ \rightsquigarrow$	
	$X \cup \{\langle G_1\ ;\ j_k^2\ ;\ G' \rangle,\ \langle G_2\ ;\ j_k^2\ ;\ G' \rangle\}$	with k fresh
(GS-Join)	$X \cup \{\underbrace{\langle j_k^x\ ;\ G \rangle, \ldots, \langle j_k^x\ ;\ G \rangle}_{x \text{ instances}}\} \ \rightsquigarrow\ X \cup \langle G \rangle$	
(GS-Choice)	$X \cup \langle (G_1 \ \oplus\ G_2)\ ;\ G' \rangle \ \rightsquigarrow$	
	$X \cup \{\langle sel_k^2\ ;\ G_1\ ;\ G' \rangle,\ \langle sel_k^2\ ;\ G_2\ ;\ G' \rangle\}$	with k fresh

Continued on the next page.

Table 2: *The semantics of global types. Continued from the previous page.*

(GS-Ch-Done)

$$X \cup \big\{\big\langle sel_k^x \;;\; G_1\big\rangle, \underbrace{\big\langle sel_k^x \;;\; G_2\big\rangle, \ldots, \big\langle sel_k^x \;;\; G_x\big\rangle}_{x-1 \text{ instances}}\big\} \;\rightsquigarrow\; X \cup \big\langle G_1\big\rangle$$

(GS-Shuffle)

$$X \cup \big\langle (G_1 \;\otimes\; G_2) \;;\; G'\big\rangle \;\rightsquigarrow\; X \cup \big\{\big\langle s\text{-}on_k^2 \;;\; G_1 \;;\; s\text{-}end_k^2 \;;\; j_k^{\,2} \;;\; G'\big\rangle, \big\langle s\text{-}on_k^2 \;;\; G_2 \;;\; s\text{-}end_k^2 \;;\; j_k^{\,2} \;;\; G'\big\rangle\big\} \qquad \text{with } k \text{ fresh}$$

(GS-Shf-Begin)

$$X \cup \big\{\big\langle s\text{-}on_k^x \;;\; G_1\big\rangle, \overbrace{\big\langle s\text{-}on_k^x \;;\; G_2\big\rangle, \ldots, \big\langle s\text{-}on_k^x \;;\; G_x\big\rangle}^{x-1 \text{ instances}}\big\} \;\rightsquigarrow\; X \cup \big\{\big\langle G_1\big\rangle, \underbrace{\big\langle s\text{-}off_k^x \;;\; G_2\big\rangle, \ldots, \big\langle s\text{-}off_k^x \;;\; G_x\big\rangle}_{x-1 \text{ instances}}\big\}$$

(GS-Shf-End)

$$X \cup \big\{\big\langle s\text{-}end_k^x \;;\; j_k^{\,x} \;;\; G_1\big\rangle, \overbrace{\big\langle s\text{-}off_k^x \;;\; G_2\big\rangle, \ldots, \big\langle s\text{-}off_k^x \;;\; G_x\big\rangle}^{x-1 \text{ instances}}\big\} \;\rightsquigarrow\; X \cup \big\{\big\langle j_k^{\,x} \;;\; G_1\big\rangle, \underbrace{\big\langle s\text{-}on_k^{x-1} \;;\; G_2\big\rangle, \ldots, \big\langle s\text{-}on_k^{x-1} \;;\; G_x\big\rangle}_{x-1 \text{ instances}}\big\}$$

(GS-Op)

$$X \cup \big\langle \big(\overset{c_2}{\underset{i=c_1}{OP}}\, G\big) \;;\; G'\big\rangle \;\rightsquigarrow\; X \cup \big\langle \big(\overset{c_2}{\underset{i=c_1}{OP}}\, G\big) \;;\; G'\big\rangle^{sp_k^x}$$

with $x = v(c_2) - v(c_1) + 1$ and k fresh, where $OP \in \{\bigoplus, \bigotimes, \|, \circledcirc\}$

(GS-Op-Nil)

$$X \cup \big\langle \big(\overset{c_2}{\underset{i=c_1}{OP}}\, G\big) \;;\; G'\big\rangle^{sp_k^x} \;\rightsquigarrow\; X \cup \big\langle G'\big\rangle \qquad \text{if } x \leq 0$$

(GS-Op-End)

$$X \cup \big\langle \big(\overset{c_2}{\underset{i=c_1}{OP}}\, G\big) \;;\; G'\big\rangle^{sp_k^x} \;\rightsquigarrow\; X \qquad \text{if } v(c_1) > v(c_2) \text{ and } x > 0$$

Continued on the next page.

Table 2: *The semantics of global types. Continued from the previous page.*

(GS-Paral-N)
$$X \cup \big\langle \big(\mathop{\|}_{i=c_1}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \rightsquigarrow X \cup \Big\{ \big\langle G[v(c_1)/i] \,;\, j_k^x \,;\, G' \big\rangle, \big\langle \big(\mathop{\|}_{i=v(c_1+1)}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \Big\} \quad \text{if } v(c_1) \leq v(c_2)$$

(GS-Shf-N)
$$X \cup \big\langle \big(\mathop{\otimes}_{i=c_1}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \rightsquigarrow X \cup \big\{ \big\langle s\text{-}on_k^x \,;\, G[v(c_1)/i] \,;\, s\text{-}end_k^x \,;\, j_k^x \,;\, G' \big\rangle, \big\langle \big(\mathop{\otimes}_{i=v(c_1+1)}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \big\} \quad \text{if } v(c_1) \leq v(c_2)$$

(GS-Choice-N)
$$X \cup \big\langle \big(\mathop{\oplus}_{i=c_1}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \rightsquigarrow X \cup \big\{ \big\langle sel_k^x \,;\, G[v(c_1)/i] \,;\, j_k^x \,;\, G' \big\rangle, \big\langle \big(\mathop{\oplus}_{i=v(c_1+1)}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \big\} \quad \text{if } v(c_1) \leq v(c_2)$$

(GS-Seq-N)
$$X \cup \big\langle \big(\mathop{\circledcirc}_{i=c_1}^{c_2} G \big) \,;\, G' \big\rangle^{sp_k^x} \rightsquigarrow X \cup \big\{ \big\langle G[v(c_1)/i] \,;\, \big(\mathop{\circledcirc}_{i=v(c_1+1)}^{c_2} G \big) \,;\, G' \big\rangle \big\} \quad \text{if } v(c_1) \leq v(c_2)$$

5. Programming Language Support

The global types defined in the previous section specify the permissible sequences of messages that participants may exchange in a given session, as well as their types. However, global types by themselves provide no implementation of the protocol. In this section, we therefore present Lang-A to write programs that embody protocols expressible in the global types of System-A. We define Lang-A such that there is almost a one-to-one correspondence between the language constructs and the syntax of local types in System-A, presented later in Section 6.

5.1. Syntax

The syntax of Lang-A is shown in Table 3. A program begins with declaring the program parameters, akin to System-A parameters. Then come message structure definitions, and the code for each actor. Both actor and message definitions can include an optional array syntax after their name. In the case of actors, this syntax declares as many of them as the array parameter. In the case of message structures, it declares as many *message types* as the array parameter. This allows the expression of protocols where both actor names and message types are parameterized, such as $\|_{i=1}^{n} a_i \xrightarrow{m_i} b_i$.

To give a taste of the language, Figure 1 shows an implementation of the sliding window protocol that we described in Section 3. The spawn statement launches n parallel instances of its block argument, one for each value of the provided index expression. Sends and receives coming from different spawned operations can be interleaved in any way possible. In this example, both the sender and the receiver spawn n parallel operations, each consisting of a repeating send/receive pair. This allows any interleaving of sends and receives, as long as no more than n sends are left unacknowledged.

```
n : param

// the sender
actor a = {
  message m : Int
  message ack : Int
  var NotDone : Boolean
  NotDone = true

  spawn(i = 1..n
    while NotDone {
      m = ...
      send(b, m) ;
      recv(b, ack) ;
      NotDone = ...
    })
}
```

```
// the receiver
actor b = {
  message m : Int
  message ack : Int
  var NotDone : Boolean
  NotDone = true

  spawn(i = 1..n
    while NotDone {
      recv(a, m) ;
      ack = ...
      send(a, ack) ;
      NotDone = ...
    })
}
```

Figure 1: *The sliding window example (p. 6) in Lang-A.*

Figure 2 shows the limited resource sharing example of page 8, formula 3 implemented in Lang-A. The program begins with the definition of parameters n and k. Then it declares n different types of lock, unlock and ack messages. This way, information about which resource is being locked is embedded in the message type itself, and the structures can be empty.

Table 3: *The syntax of Lang-A. Literals are underlined.*

$\mathscr{P}$	::=	ParamDecl* StructDef* ActorDef*	
ParamDecl	::=	param n	$n \in$ ParamNames
StructDef	::=	struct sn ([Const])? = { VarDecl* }	$sn \in$ StructNames
ActorDef	::=	actor pn ([Const])? = { Stmt }	$pn \in$ ActorNames
Stmt	::=	Spawn | If | While | Repeat | For | Shuffle	
		| Select | VarDecl | SimpleStmt | Stmt ; Stmt	
		| skip	
VarDecl	::=	var w : sort	$w \in$ VariableNames
		| var ar : sort [Const]	$ar \in$ ArrayNames
		| message mn : sort ([Const | ind])?	$mn \in$ MsgNames
			$ind \in$ IndexNames
Spawn	::=	spawn (IndexDecl Block)	
		| spawn (Block Block)	
IndexDecl	::=	ind = Const .. Const	$ind \in$ IndexNames
If	::=	if BooleanExpr Block else Block	
While	::=	while BooleanExpr Block	
Repeat	::=	repeat Const Block	
For	::=	for IndexDecl Block	
Shuffle	::=	shuffle (IndexDecl Block)	
		| shuffle (Block Block)	
Select	::=	select (IndexDecl Block)	
		| select (Block Block)	
SimpleStmt	::=	*assignments, arithmetic operations etc.*	
		| send (pn , mn)	$pn \in$ ActorNames
		| recv (pn , mn)	$mn \in$ MsgNames
		| send (pn [IndexExp] , mn)	$ind \in$ IndexNames
		| recv (pn [IndexExp] , mn)	
Nat	::=	0 | 1 | 2 | ...	
Const	::=	Nat | n	$n \in$ ParamNames
IndexExp	::=	Const | ind	$ind \in$ IndexNames
BooleanExpr	::=	*comparisons etc.*	
Block	::=	{ Stmt }	
sort	::=	*struct name, or primitive type*	

The server spawns n operations to deal with the simultaneous locking of n different resources. The index declared in the spawn statement identifies the message used by the particular instance. The server can handle any of the k clients (indexed by j), as long as the current client is only locking–unlocking one resource (indexed by i). The select statement can only be used if the first communication statement that follows is a receive. The form used here includes an index expression j = 1..n in the beginning, and this index is used to match either the sort of the received message, or the sender. In this example, j determines the client that sent the request. Even though each client spawns one lock operation per resource, these do not cause inconsistencies because there is only one parallel handler per resource on the server (indexed by i). This way, the server serializes the requests, so that the same resource cannot be locked by more than one client at the same time. Only after the receipt of the relevant unlock message (indexed by i) can the same resource be locked again by another (or the same) client.

```
param n
param k
struct lock[n] = { }
struct unlock[n] = { }
struct ack[n] = { }

actor s = {
  var NotDone : Boolean
  NotDone = true

  spawn(i = 1..n
    message l : lock[i]
    message a : ack[i]
    message u : unlock[i]
    while NotDone {
      select(j = 1..k
        recv(c[j], l) ;
        send(c[j], a) ;
        recv(c[j], u) ;)
      NotDone = ...
    })
}
```

```
actor c[k] = {
  var NotDone : Boolean
  NotDone = true

  spawn(i = 1..n
    while NotDone {
      message l : lock[i]
      message a : ack[i]
      message u : unlock[i]
      send(s, l) ;
      recv(s, a) ;
      // compute ... ;
      send(s, u) ;
      NotDone = ...
    })
}
```

Figure 2: *The resource sharing example (p. 7) in Lang-A.*

5.2. Semantics and Trace Generation

We give the operational semantics of Lang-A in Table 4. The purpose of the semantics is to explain what traces a program can generate when executing, which will play a central role in the correctness proof of Section 10.

The semantics employ configurations of the form $(SEL, SHUF, T, M, V, \Pi, R)$. SEL and $SHUF$ are partial functions from markers to integers that we explain later; T is the trace produced so far; M is the multiset of pending (sent but not received) messages; V holds the values of variables, constants and parameters; Π is the multiset of currently running actors; and R is the program code.

Definition 4 (Lang-A Traces)**.** The set $tr\,(P)$ of traces of a program $P \in \mathscr{P}$ consists of all traces that can be derived by applying the semantic rules in Table 4 to the initial configuration $\big(\bot, \bot, \epsilon, \emptyset, \emptyset, \emptyset, P\big)$ until termination. That is, all different values T can have when P terminates, when T's initial value is set to ϵ. We say that P terminates when the set Π in the configuration contains nothing but $\big\langle \mathsf{skip} \big\rangle$ processes.

Only the first two rules in Table 4 rewrite the program code: (PL-ParInit) parses a parameter declaration and stores its value in V, assuming the value is input by the user at that point. (PL-ActorInit) parses an actor declaration, and places a new process in Π with the given behavior. It also appends skip to the code, to enable termination; this is similar to the appending of τ to global types by (GS-Init) in Table 2. An actor a with behavior $B \in \text{Block} \cup \text{Stmt}$ is written $\langle B \rangle_a$. As in the semantics of global types, rules only mention those parts of the configuration that they alter. For example T is omitted everywhere except for (PL-Recv). Likewise, we omit braces around singleton sets.

We use the customary rules for loops, conditionals, and sequential composition. As in the semantics of global types, we simplify the rules by relying on an auxiliary evaluation function $v(\cdot)$ that subsumes the standard reduction rules for computations in procedural languages [33, 45]. As before, we assume v to be side-effect free, as it simply consults the store V and returns the value of the expression provided as its argument. The store V is updated by (PL-Assign).

Sends are asynchronous. The respective rule (PL-Send) places the sent message in M, recording its type in the superscript, while the subscript stores the sender and the receiver. The only rule allowing code prefixed with a receive to progress is (PL-Recv). Hence, receives are blocking, which enables us to enforce message sequencing. Note that this is also the only rule that updates the trace T, which means that event production happens only on receives. In effect, communication is asynchronous; the order of events belonging to multiple sends or multiple shuffled receives is non-deterministic.

Note that since Π is a multiset, it may contain many actors with the same name. In this context, rather than using the term *actor* to refer to $\langle B \rangle_a \in \Pi$, we henceforth use the term *process*. We make this distinction due to the way the semantics handle choice, shuffling and concurrent composition: a separate process is spawned for each branch B_i, as in $\langle B_1 \rangle_a, \ldots, \langle B_n \rangle_a$, all of which correspond to the same actor a, denoted in the subscript. Consequently, Π is the set of currently running processes, which we assume to run concurrently. In cases where many of them can receive a message in M, a non-deterministic choice is made by rule (PL-Recv). This is a general assumption in our system: whenever more than one reduction step can be taken, a non-deterministic choice is made as to which rule to apply, or which concurrent process to reduce.

Select statements choose between the different branches of the behavior of an actor a, depending on the next received message. To implement the statement, the semantics create a concurrent process for each possible branch, all corresponding to the same actor a. All concurrent processes block waiting to receive a message, and when one of them does, it determines the branch to be followed. At this point, all but the process corresponding to the branch taken are discarded.

Branches must be tracked across statement sequences, nested choices, and actors created through spawn statements. To achieve this, the semantics associate each encountered select statement with a unique identifier (PL-Select). For each different branch, the respective process has a *sel* function, written as a superscript. This function maps the identifier introduced by (PL-Select) to the number of the branch that the process represents. Correspondingly, the configuration employs the partial function *SEL* to record the choice once it is made. A choice is made by the first process that receives a message, because the sort of the message as well as its sender determine the branch to be taken. Hence the *SEL* mapping is updated by the first branch to execute rule (PL-Recv).

The idea is that a concurrent process is only allowed to progress if its local *sel* mapping matches that of the *SEL* function on the configuration level. In this fashion, (PL-Recv) only allows a branch to progress in two cases: either the value of *SEL* is undefined – in which case it is updated to reflect the local one; or it is already the same as *sel*, meaning that the process is on the same branch as the one that set the value beforehand. Branches that do not satisfy any of these conditions are disposed of in rule (PL-SelectDone).

Note that concurrent processes created through a spawn statement all inherit their parent's *sel* map (PL-Spawn); hence, either all of them can progress (PL-Recv), of none (PL-SelectDone).

Shuffles are implemented in a similar way, using *SHUF* on the configuration level and *shuf* on the process level. The difference is that when a branch finishes, we need to allow one of the others to execute (because they all need to execute, albeit atomically as in the global type semantics). This is accomplished by introducing a special marker u_k when spawning the shuffle branches (PL-Shuffle). This resets the value of *SHUF* to $\bot$ when the branch completes (PL-ShuffleDone), allowing the next branch to execute (PL-Recv).

Table 4: *The semantics of Lang-A. The rules transform configurations* $(SEL, SHUF, T, M, V, \Pi, R)$*, where SEL and SHUF are partial functions from markers to values; T is the trace produced so far; M is the multiset of pending messages; V the value store;* Π *the multiset of executing actors* $\langle B \rangle_a$ *such that a is the actor executing behavior* $B \in \mathrm{Stmt}$*; and R is the program code. Like before, the rules omit unchanged elements. Function* $v(\cdot)$ *evaluates expressions without side-effects;* $[\cdot/\cdot]$ *denotes substitution of free variables;* $[\cdot \mapsto \cdot]$ *updates functions point-wise; the marker* u_k *is used to implement shuffling. Table 3 defines the remaining symbols.*

(PL-ParInit)	$(V,\ \mathsf{param}\ n \cdot R) \ \rightsquigarrow\ \big(V \cup \{(n, v(n))\},\ R\big)$	
(PL-ActorInit)	$(\Pi,\ \mathsf{actor}\ a = \{B\} \cdot R) \ \rightsquigarrow\ (\Pi \cup \langle B; \mathsf{skip}\rangle_a^{\bot,\bot},\ R)$	
(PL-Empty)	$\mathsf{skip}\ ;\ B \rightsquigarrow B$	
(PL-Assign)	$\big(V,\ \Pi \cup \langle w = exp\ ;\ B\rangle_a^{sel,\ shuf}\big) \ \rightsquigarrow\ \big(updated(V, w, v(exp)),\ A \cup \langle B\rangle_a^{sel,\ shuf}\big)$	
(PL-Seq)	$B_1\ ;\ B_2 \rightsquigarrow B_1'\ ;\ B_2$	if $B_1 \rightsquigarrow B_1'$
(PL-IfTrue)	$\mathsf{if}\ exp\ B_1\ \mathsf{else}\ B_2\ ;\ B' \rightsquigarrow B_1\ ;\ B'$	if $v(exp) = \mathsf{true}$
(PL-IfFalse)	$\mathsf{if}\ exp\ B_1\ \mathsf{else}\ B_2\ ;\ B' \rightsquigarrow B_2\ ;\ B'$	if $v(exp) = \mathsf{false}$
(PL-For)	$\mathsf{for}\ i = c_1..c_2\ B \rightsquigarrow B\big[v(c_1)/i\big]\ ;\ \mathsf{for}\ i = v(c_1+1)..c_2\ B$	if $v(c_1) \le v(c_2)$

Continued on the next page.

Table 4: *The semantics of Lang-A – Continued from the previous page.*

(PL-ForNil) $\mathsf{for}\ i = c_1..c_2\ B \rightsquigarrow \mathsf{skip}$ if $v(c_1) > v(c_2)$

(PL-Repeat) $\mathsf{repeat}\ c\ B \rightsquigarrow \mathsf{for}\ i = 1..c\ B$

(PL-While) $\mathsf{while}\ exp\ B \rightsquigarrow B\ ;\ \mathsf{while}\ exp\ B$ if $v(exp) = \mathsf{true}$

(PL-WhileNil) $\mathsf{while}\ exp\ B \rightsquigarrow \mathsf{skip}$ if $v(exp) = \mathsf{false}$

(PL-Aux)
$$\Pi \cup \left\langle f(i = c_1..c_2\ B)\ ;\ B' \right\rangle_a^{sel,\ shuf} \rightsquigarrow \Pi \cup \left\langle f(i = c_1..c_2\ B)\ ;\ B' \right\rangle_a^{sel,\ shuf,\ sp_k^x}$$
for $x = v(c_2) - v(c_1) + 1$ and k fresh, where $f \in \{\mathsf{select},\ \mathsf{shuffle},\ \mathsf{spawn}\}$

(PL-AuxEnd) $\Pi \cup \left\{ \left\langle f(i = c_1..c_2\ B)\ ;\ B' \right\rangle_a^{sel,\ shuf,\ sp_k^x} \right\} \rightsquigarrow \Pi$
if $v(c_1) > v(c_2)$ and $x > 0$ where $f \in \{\mathsf{select},\ \mathsf{shuffle},\ \mathsf{spawn}\}$

(PL-AuxNil) $\Pi \cup \left\langle f(i = c_1..c_2\ B)\ ;\ B' \right\rangle_a^{sel,\ shuf,\ sp_k^x} \rightsquigarrow \Pi \cup \left\langle B' \right\rangle_a^{sel,\ shuf}$
if $x \leq 0$, where $f \in \{\mathsf{select},\ \mathsf{shuffle},\ \mathsf{spawn}\}$

(PL-Spawn) $\Pi \cup \left\langle \mathsf{spawn}\ (B_1\ B_2)\ ;\ B_3 \right\rangle_a^{sel,\ shuf} \rightsquigarrow$
$\Pi \cup \left\{ \left\langle B_1\ ;\ j_k^2\ ;\ B_3 \right\rangle_a^{sel,\ shuf}, \left\langle B_2\ ;\ j_k^2\ ;\ B_3 \right\rangle_a^{sel,\ shuf} \right\}$ for k fresh

(PL-SpawnN)
$$\left(\Pi \cup \left\{ \left\langle \mathsf{spawn}\ (i = c_1..c_2\ \{B\})\ ;\ B' \right\rangle_a^{sel,\ shuf,\ sp_k^x} \right\}\right) \rightsquigarrow$$
$$\left(\Pi \cup \left\{ \left\langle B[v(c_1)/i]\ ;\ j_k^x\ ;\ B' \right\rangle_a^{sel,\ shuf}, \left\langle \mathsf{spawn}\ (i = v(c_1+1)..c_2\ B)\ ;\ B' \right\rangle_a^{sel,\ shuf,\ sp_k^x} \right\}\right)$$
if $v(c_1) \leq v(c_2)$

(PL-Join) $\Pi \cup \underbrace{\left\{ \left\langle j_k^x\ ;\ B \right\rangle_a^{sel,\ shuf}, \ldots, \left\langle j_k^x\ ;\ B \right\rangle_a^{sel,\ shuf} \right\}}_{x \text{ instances}} \rightsquigarrow \Pi \cup \left\langle B \right\rangle_a^{sel,\ shuf}$

Continued on the next page.

Table 4: *The semantics of Lang-A – Continued from the previous page.*

(PL-Send) $\big(M,\, \Pi \cup \big\langle \mathsf{send}(b, m)\ ;\ B \big\rangle_a^{sel,\, shuf}\big) \ \rightsquigarrow$

$$\big(M \cup (val)_{a,b}^{t},\, \Pi \cup \big\langle B \big\rangle_a^{sel,\, shuf}\big) \quad \text{if } typeof(m) = t \text{ and } v(m) = val$$

(PL-Recv)

$$\big(SEL,\, SHUF,\, T,\, M \cup (val)_{a,b}^{t},\, \Pi \cup \big\langle \mathsf{recv}(a, m)\ ;\ B \big\rangle_b^{sel,\, shuf}\big) \ \rightsquigarrow$$
$$\big(SEL[x \mapsto sel(x) \,|\, x \in dom(sel)],\ SHUF[x \mapsto shuf(x) \,|\, x \in dom(shuf)],$$
$$T \cdot (a \xrightarrow{t} b),\, M,\, \Pi \cup \big\langle B[val/m] \big\rangle_b^{sel,\, shuf}\big)$$
$$\text{if } typeof(m) = t \text{ and}$$
$$SEL(x) \in \{\bot,\, sel(x)\}\ \forall x \in dom(sel)$$
$$SHUF(y) \in \{\bot,\, shuf(y)\}\ \forall y \in dom(shuf)$$

(PL-Select) $\Pi \cup \big\langle \mathsf{select}(B_1\, B_2)\ ;\ B_3 \big\rangle_a^{sel,\, shuf} \ \rightsquigarrow$

$$\Pi \cup \big\{ \big\langle B_1\ ;\ B_3 \big\rangle_a^{sel[k \mapsto 1],\, shuf} \big\langle B_2\ ;\ B_3 \big\rangle_a^{sel[k \mapsto 2],\, shuf} \big\} \qquad \text{for } k \text{ fresh}$$

(PL-SelectDone) $\big(SEL,\, \Pi \cup \big\langle B \big\rangle_a^{sel,\, shuf}\big) \ \rightsquigarrow \ \big(SEL,\, \Pi\big)$

$$\text{if for an } x \in dom(sel) : SEL(x) \notin \{\bot,\, sel(x)\}$$

(PL-SelectN) $\Pi \cup \big\{ \big\langle \mathsf{select}(i = c_1..c_2\, \{B\})\ ;\ B' \big\rangle_a^{sel,\, shuf,\, sp_k^x} \big\} \ \rightsquigarrow$

$$\Pi \cup \big\{ \big\langle B[v(c_1)/i]\ ;\ B' \big\rangle_a^{sel[k \mapsto v(c_1)],\, shuf},$$
$$\big\langle \mathsf{select}(i = v(c_1 + 1)..c_2\, B)\ ;\ B' \big\rangle_a^{sel,\, shuf,\, sp_k^x} \big\}$$
$$\text{if } v(c_1) \le v(c_2)$$

(PL-Shuffle) $\Pi \cup \big\langle \mathsf{shuffle}(B_1, B_2)\ ;\ B_3 \big\rangle_a^{sel,\, shuf} \ \rightsquigarrow$

$$\Pi \cup \big\{ \big\langle B_1\ ;\ u_k\ ;\ j_k^2\ ;\ B_3 \big\rangle_a^{sel,\, shuf[k \mapsto 1]},$$
$$\big\langle B_2\ ;\ u_k\ ;\ j_k^2\ ;\ B_3 \big\rangle_a^{sel,\, shuf[k \mapsto 2]} \big\} \qquad \text{for } k \text{ fresh}$$

(PL-ShuffleDone)

$$\big(SHUF,\, \Pi \cup \big\{ \big\langle u_k\ ;\ j_k^x\ ;\ B \big\rangle_a^{sel,\, shuf} \big\}\big) \ \rightsquigarrow \ SHUF[k \mapsto \bot],\, \Pi \cup \big\langle j_k^x\ ;\ B \big\rangle_a^{sel,\, shuf[k \mapsto \bot]}$$

Continued on the next page.

Table 4: *The semantics of Lang-A – Continued from the previous page.*

$$\text{(PL-ShuffleN)}\quad \Pi \cup \left\{ \left\langle \mathsf{shuffle}(i = c_1..c_2\, B)\ ;\ B' \right\rangle_a^{sel,\, shuf,\, sp_k^x} \right\} \rightsquigarrow$$
$$\Pi \cup \left\{ \left\langle B[v(c_1)/i]\ ;\ u_k\ ;\ j_k^x\ ;\ B' \right\rangle_a^{sel,\, shuf[k \mapsto v(c_1)]},\right.$$
$$\left. \left\langle \mathsf{shuffle}(i = v(c_1+1)..c_2\, B)\ ;\ B' \right\rangle_a^{sel,\, shuf,\, sp_k^x} \right\}$$
$$\text{if } v(c_1) \leq v(c_2)$$

6. Local Types

A local type specifies the abstract behavior of a single protocol participant, for example of one of the actors in a Lang-A program. Furthermore, local types specify the behavior restrictions that a global type implies for each protocol participant. This section defines local types and shows how to infer them from Lang-A code. Later, we discuss how local types can be used to check whether a Lang-A program conforms to a global type.

The syntactic category $\mathscr{L}$ of local types is defined by the grammar in Table 5. We will use the variable $L \in \mathscr{L}$, often indexed, to refer to local types. In the grammar,

(L-Send) denotes sending a message of type t to actor a.

(L-Recv) denotes receiving a message of type t from actor a.

(L-Seq), (L-Choice), (L-Shuffle), (L-Exp), (L-KleeneStar) describe the same concepts as in the global types.

(L-Paral) is also defined as in the case of global types. Like there, the local type $(a!t\ ;\ a!u) \parallel a?v$ allows three orderings of the events $T = a!t$, $U = a!u$ and $V = a?v$, namely TUV, TVU, and VTU. As above, the specification $a!t\ ;\ a!u$ enforces that T happens before U.

6.1. Local Type Inference

This section explains how local types can be derived from the code of a Lang-A program using the inference rules in Table 6. To clarify the presentation, we distinguish between the types of program constructs (such as

Table 5: *The syntax of local types. As in the syntax of global types, the grammar contains the auxiliary symbols* $i \in \mathit{IndexNames}$; $n, n_1, n_2 \in \mathit{ParamNames} \cup \mathbb{N}$; $a \in \mathit{ActorNames} \cup \{\alpha_j \mid \alpha \in \mathit{ActorNames},\ j \in \mathit{IndexNames}\}$; *and* $t \in \mathit{MsgNames} \cup \{\mu_j \mid \mu \in \mathit{MsgNames},\ j \in \mathit{IndexNames}\}$.

$\mathscr{L} ::=$	$(\mathscr{L})$	(L-Paren)	$\mid$	τ	(L-Empty)
$\mid$	$a!t$	(L-Send)	$\mid$	$a?t$	(L-Recv)
$\mid$	$\mathscr{L}\ ;\ \mathscr{L}$	(L-Seq)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\circledcirc}} \mathscr{L}_i$	(L-Seq-N)
$\mid$	$\mathscr{L} \oplus \mathscr{L}$	(L-Choice)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\bigoplus}} \mathscr{L}_i$	(L-Choice-N)
$\mid$	$\mathscr{L} \parallel \mathscr{L}$	(L-Paral)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\Vert}} \mathscr{L}_i$	(L-Paral-N)
$\mid$	$\mathscr{L} \otimes \mathscr{L}$	(L-Shuffle)	$\mid$	$\overset{n_2}{\underset{i=n_1}{\bigotimes}} \mathscr{L}_i$	(L-Shuffle-N)
$\mid$	$\mathscr{L}^n$	(L-Exp)	$\mid$	$\mathscr{L}^*$	(L-KleeneStar)

messages or variables) and types in System-A (which describe communication protocols). We use the term *sort* for the former, while the term *type* is used exclusively to refer to System-A local types.

By S we denote the environment that contains the sorts of variables that appear in a Lang-A program. S contains statements of the form $w : t$ when w is a message of sort t; $w : \mathit{const}$ when w is a constant (numerical constant or parameter name), and $w : \mathit{ind}$ when w is the name of an index. For messages, t can have the form $m[i]$ with $S \vdash i : \mathit{ind}$ (that is, i is a valid index). The sort of expressions (e.g. boolean expressions) is also determined through S. The environment Π is used to resolve actor names. It supplies judgments of the form $x : \mathit{actor}$, meaning that x is a valid actor name, including the case $x = a[i]$ with $S \vdash i : \mathit{ind}$. The rules for updating S and Π are standard: S is updated on message, parameter and index declarations, as well as when encountering arithmetic and boolean expressions in the program. Π is updated based on actor definitions. We omit the formal definition of these rules for the sake of brevity.

The judgments presented in Table 6 are of the form $\Gamma \vdash R : L$, read "under the environment Γ, the sequence of actions R has (local) type L". The result of the typing is given by rule (Inf-Environment), which produces an environment Δ containing all the local types in the program.

Table 6: *Rules for inferring local types from Lang-A programs. The environments S and Π contain sorts of variables and actor names respectively. Γ assigns local types $L \in \mathscr{L}$ to (sequences of) actions $R \in \mathrm{Stmt}$. P is the program being typed.*

(Inf-Skip)

$$\frac{}{\Gamma \vdash \mathsf{skip} : \tau}$$

(Inf-Compute)

$$\frac{}{\Gamma \vdash \mathrm{Computation} : \tau}$$

(Inf-Send)

$$\frac{\Pi \vdash a : \mathit{actor} \qquad S \vdash x : m}{\Gamma \vdash \mathsf{send}(a, x) : a!m}$$

(Inf-Recv)

$$\frac{\Pi \vdash a : \mathit{actor} \qquad S \vdash x : m}{\Gamma \vdash \mathsf{recv}(a, x) : a?m}$$

(Inf-Seq)

$$\frac{\Gamma \vdash R_1 : L_1 \qquad \Gamma \vdash R_2 : L_2}{\Gamma \vdash R_1; R_2 : (L_1; L_2)}$$

(Inf-For)

$$\frac{\Gamma \vdash R[i/k] : L_i \qquad k \in \mathit{fv}(R) \qquad S \vdash c_1 : \mathit{cons} \qquad S \vdash c_2 : \mathit{cons}}{\Gamma \vdash \mathsf{for}\ k = c_1..c_2\ R : \left(\bigcirc_{i=c_1}^{c_2} L_i\right)}$$

(Inf-Spawn-N)

$$\frac{S \vdash c_1 : \mathit{cons} \qquad S \vdash c_2 : \mathit{cons} \qquad \Gamma \vdash R[i/k] : L_i \qquad k \in \mathit{fv}(R)}{\Gamma \vdash \mathsf{spawn}(k = c_1..c_2\ R) : \left(\parallel_{i=c_1}^{c_2} L_i\right)}$$

(Inf-Spawn)

$$\frac{\Gamma \vdash R_1 : L_1 \qquad \Gamma \vdash R_2 : L_2}{\Gamma \vdash \mathsf{spawn}(R_1\ R_2) : (L_1 \parallel L_2)}$$

(Inf-Shuffle)

$$\frac{\Gamma \vdash R_1 : L_1 \qquad \Gamma \vdash R_2 : L_2 \qquad \mathit{first}\,(L_1) = a?m \qquad \mathit{first}\,(L_2) = a'?m' \qquad a \neq a' \vee m \neq m'}{\Gamma \vdash \mathsf{shuffle}(R_1\ R_2) : (L_1 \otimes L_2)}$$

(Inf-Shuffle-N)

$$\frac{S \vdash c_1 : \mathit{cons} \qquad S \vdash c_2 : \mathit{cons} \qquad \Gamma \vdash R[i/k] : L_i \qquad k \in \mathit{fv}(R) \qquad \mathit{first}\,(L_i) = x?y \qquad x = a_i \vee y = m_i}{\Gamma \vdash \mathsf{shuffle}(k = c_1..c_2\ R) : \left(\bigotimes_{i=c_1}^{c_2} L_i\right)}$$

Continued on the next page.

Table 6: *Local type inference – continued from the previous page.*

(Inf-Select-N)

$$\frac{S \vdash c_1 : \mathit{cons} \qquad S \vdash c_2 : \mathit{cons} \qquad \Gamma \vdash R[i/k] : L_i \qquad k \in \mathit{fv}(R) \qquad \mathit{first}\,(L_i) = x?y \qquad x = a_i \vee y = m_i}{\Gamma \vdash \mathsf{select}(k = c_1..c_2\ R) : \big(\bigoplus_{i=c_1}^{c_2} L_i\big)}$$

(Inf-Select)

$$\frac{\Gamma \vdash R_1 : L_1 \qquad \Gamma \vdash R_2 : L_2 \qquad \mathit{first}\,(L_1) = a?m \qquad \mathit{first}\,(L_2) = a'?m' \qquad a \neq a' \vee m \neq m'}{\Gamma \vdash \mathsf{select}(R_1\ R_2) : (L_1 \oplus L_2)}$$

(Inf-Repeat)

$$\frac{\Gamma \vdash R : L \qquad S \vdash \mathsf{n} : \mathit{cons}}{\Gamma \vdash \mathsf{repeat}\ n\ R : (L^n)}$$

(Inf-While)

$$\frac{\Gamma \vdash R : L \qquad S \vdash \mathsf{exp} : \mathit{Boolean}}{\Gamma \vdash \mathsf{while\ exp}\ R : (L^*)}$$

(Inf-If)

$$\frac{\Gamma \vdash R_1 : L_1 \qquad \Gamma \vdash R_2 : L_2 \qquad \mathit{first}\,(L_1) = a!m \qquad \mathit{first}\,(L_2) = a'!m' \qquad a \neq a' \vee m \neq m' \qquad S \vdash \mathsf{exp} : \mathit{Boolean}}{\Gamma \vdash \mathsf{if\ exp}\ R_1\ \mathsf{else}\ R_2 : (L_1 \oplus L_2)}$$

(Inf-Environment)

$$\frac{\Pi = \{a \,|\, \text{“}\mathsf{actor}\ a = \{R_a\}\text{”} \in P\} \qquad \forall a \in \Pi,\ \Gamma \vdash R_a : L_a}{\Gamma \vdash P : \Delta \text{ where } \Delta = \{a : L_a \,|\, a \in \Pi\},}$$

Note that computations (assignments, arithmetic operations etc.) type to τ, since they are not observable actions (Inf-Compute). Here, “computation” refers to the first line of rule SimpleStmt in Table 3. While most of the other rules are straightforward, the ones involving shuffling and choice require some discussion. The intended meaning of shuffling is to support the reordering of its arguments. Implementing this in a traditional actor language would mean that the arguments of the shuffling operator would translate into different message handlers, so that the execution order is determined by the order in which messages are received. Hence, the first communication primitive (given by function $\mathit{first}\,(\cdot)$) in the arguments of a shuffle statement should be a receive. This is enforced by (Inf-Shuffle), which also demands that receive statements from different arguments be distinguishable either because the sender is different, or because the expected message's sort is different. Rule (Inf-Shuffle-N) achieves the same by making sure that the index appears free in the argument, that is, $k \in \mathit{fv}(R)$, and that it is part of the sender's name or part of the expected message sort. The same premises apply to the cases of (Inf-Select) and (Inf-Select-N), respectively.

Observe that the side condition is reversed in the case of (Inf-If), where the branches have to start with a send statement. While select and shuffle statements are an internal choice mechanism, if statements implement external choice.

7. Projection

Besides the characterization of actor behavior defined in the previous section, local types also specify the behavior restrictions that a global type implies for each participant.

For example, in the sliding window protocol of Section 3, the permissible behaviors of the sender a are described by the local type $\|_{i=1}^{n}(b!m\ ;\ b?\text{ack})^*$. Temporarily ignoring the leading $\|_{i=1}^{n}$ symbol, the local type is $(b!m\ ;\ b?\text{ack})^*$, which means sending a message and then receiving an acknowledgment an unbounded number of times. Assuming that the window size n is a parameter, any interleaving of n such sequences is possible, with the obvious constraint of not receiving more acknowledgments than the number of messages sent. This is ensured by composing sequences of the form $(b!m\ ;\ b?\text{ack})$, where ordering is forced by the ; operator.

The above example local type is a projection of the global type of the sliding window protocol onto the sender actor. The projection function $\triangleright$ is formally defined in Table 7; $G \triangleright p$ is read "the projection of global type G onto the participant p" and the result is a local type. Applying the projection function to all participants in G gives an *environment* $\Delta = \{p : L_p\}_{p\in\Pi}$, which maps actors to local types. In the table, the last four projection rules are only defined for $n_1 \leq k \leq n_2$ because we assume that we are projecting onto an actor in the range of the operator. As before, $R[x/y]$ denotes the substitution of the free instances of y in R by x. Also, more specific rules take precedence over less specific ones. For example, (P-Paral-N-Ind) takes precedence over (P-Paral-N) when projecting onto an indexed actor. In (P-Interaction), actor names are considered equal only if they match completely, that is, including indices.

Table 7: *The projection function for deriving the local type of a participant p from a global type $G \in \mathscr{G}$. It is $p \in ActorNames \cup \{\alpha_j \mid \alpha \in ActorNames,\, j \in IndexNames\}$.*

$$
\begin{array}{rcll}
(a \xrightarrow{m} b) \triangleright p & ::= & \begin{cases} b!m & \text{if } p = a \neq b \\ a?m & \text{if } p = b \neq a \\ a!m\,;\,a?m & \text{if } p = a = b \\ \tau & \text{otherwise} \end{cases} & \text{(P-Interaction)} \\
G^n \triangleright p & ::= & (G \triangleright p)^n & \text{(P-Exp)} \\
(G_1 \oplus G_2) \triangleright p & ::= & (G_1 \triangleright p) \oplus (G_2 \triangleright p) & \text{(P-Choice)} \\
(G_1 \parallel G_2) \triangleright p & ::= & (G_1 \triangleright p) \parallel (G_2 \triangleright p) & \text{(P-Paral)} \\
(G_1\,;\,G_2) \triangleright p & ::= & (G_1 \triangleright p)\,;\,(G_2 \triangleright p) & \text{(P-Seq)} \\
(G_1 \otimes G_2) \triangleright p & ::= & (G_1 \triangleright p) \otimes (G_2 \triangleright p) & \text{(P-Shuffle)} \\
(\mathop{\circledcirc}_{i=n_1}^{n_2} G_i) \triangleright p & ::= & \mathop{\circledcirc}_{i=n_1}^{n_2} (G_i \triangleright p) & \text{(P-Seq-N)} \\
(\bigoplus_{i=n_1}^{n_2} G_i) \triangleright p & ::= & \bigoplus_{i=n_1}^{n_2} (G_i \triangleright p) & \text{(P-Choice-N)} \\
(\bigotimes_{i=n_1}^{n_2} G_i) \triangleright p & ::= & \bigotimes_{i=n_1}^{n_2} (G_i \triangleright p) & \text{(P-Shuffle-N)} \\
(\mathop{\Vert}_{i=n_1}^{n_2} G_i) \triangleright p & ::= & \mathop{\Vert}_{i=n_1}^{n_2} (G_i \triangleright p) & \text{(P-Paral-N)} \\
(\mathop{\circledcirc}_{i=n_1}^{n_2} G_i) \triangleright p_k & ::= & \begin{cases} G_i[k/i] \triangleright p_k & \text{if } p_k[i/k] \in G_i \\ \mathop{\circledcirc}_{i=n_1}^{n_2} (G_i \triangleright p_k) & \text{otherwise} \end{cases} & \text{(P-Seq-N-Ind)} \\
(\bigoplus_{i=n_1}^{n_2} G_i) \triangleright p_k & ::= & \begin{cases} G_i[k/i] \triangleright p_k & \text{if } p_k[i/k] \in G_i \\ \bigoplus_{i=n_1}^{n_2} (G_i \triangleright p_k) & \text{otherwise} \end{cases} & \text{(P-Choice-N-Ind)} \\
(\bigotimes_{i=n_1}^{n_2} G_i) \triangleright p_k & ::= & \begin{cases} G_i[k/i] \triangleright p_k & \text{if } p_k[i/k] \in G_i \\ \bigotimes_{i=n_1}^{n_2} (G_i \triangleright p_k) & \text{otherwise} \end{cases} & \text{(P-Shuffle-N-Ind)} \\
(\mathop{\Vert}_{i=n_1}^{n_2} G_i) \triangleright p_k & ::= & \begin{cases} G_i[k/i] \triangleright p_k & \text{if } p_k[i/k] \in G_i \\ \mathop{\Vert}_{i=n_1}^{n_2} (G_i \triangleright p_k) & \text{otherwise} \end{cases} & \text{(P-Paral-N-Ind)}
\end{array}
$$

For the lock/unlock example of Section 3, projecting the global type G of formula 2 (p. 7) onto a client c_k (with $1 \leq k \leq n$) and the server s yields

$$
\begin{aligned}
L_k &= G \triangleright c_k \\
&= \bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_i \;;\; c_i \xrightarrow{unlock} s) \triangleright c_k \\
&= (c_k \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_k \;;\; c_k \xrightarrow{unlock} s) \triangleright c_k && \text{(P-Shuffle-N-Ind)} \\
&= (c_k \xrightarrow{lock} s) \triangleright c_k \;;\; (s \xrightarrow{ack} c_k) \triangleright c_k \;;\; (c_k \xrightarrow{unlock} s) \triangleright c_k && \text{(P-Seq)} \\
&= s!lock \;;\; s?ack \;;\; s!unlock && \text{(P-Interaction)}
\end{aligned}
$$

$$
\begin{aligned}
L_s &= G \triangleright s \\
&= \bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_i \;;\; c_i \xrightarrow{unlock} s) \triangleright s \\
&= \bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \;;\; s \xrightarrow{ack} c_i \;;\; c_i \xrightarrow{unlock} s \triangleright s) && \text{(P-Shuffle-N)} \\
&= \bigotimes_{i=1}^{n} (c_i?lock \;;\; c_i!ack \;;\; c_i?unlock)\,. && \text{(P-Interaction), (P-Seq)}
\end{aligned}
$$

Similarly, the projected local types for the server s and a client c_m (with $1 \leq m \leq k$) of the resource sharing protocol (formula 3, p. 8) are

$$
\begin{aligned}
L_s &= \prod_{i=1}^{n} \Big(\bigoplus_{j=1}^{k} (c_j?lock_i \;;\; c_j!ack_i \;;\; c_j?unlock_i) \Big)^*, \\
L_{c_m} &= \prod_{i=1}^{n} \big(s!lock_i \;;\; s?ack_i \;;\; s!unlock_i \big)^*.
\end{aligned}
$$

In order for the last four rules of Table 7 to work properly, we disallow global types where the same name appears with more than one index, each introduced by a different operator application. An example of an unsupported case would be the following:

$$
\prod_{i=v_1}^{n_1} \prod_{j=v_1}^{n_2} a_i \xrightarrow{m} a_j
$$

The problem appears because the two indices i and j come from two different applications of the concurrent composition operator, hence the result of the

Table 8: *Rules for eliminating τ terms from local types.*

$L \,;\, \tau$	$\longrightarrow_{norm}$	L	$\tau \,;\, L$	$\longrightarrow_{norm}$	L
$L \oplus \tau$	$\longrightarrow_{norm}$	L	$\tau \oplus L$	$\longrightarrow_{norm}$	L
$L \otimes \tau$	$\longrightarrow_{norm}$	L	$\tau \otimes L$	$\longrightarrow_{norm}$	L
$L \parallel \tau$	$\longrightarrow_{norm}$	L	$\tau \parallel L$	$\longrightarrow_{norm}$	L
τ^n	$\longrightarrow_{norm}$	τ	τ^*	$\longrightarrow_{norm}$	τ

projection onto a participant a_k depends on where the value of k lies in the sub-intervals defined by v_1, v_2, n_1 and n_2. Since our projection function does not take external restrictions into account, we omit the treatment of such cases in this article, and discuss a possible solution in Section 11.

Finally, for reasons of presentation clarity, we assume a hygiene condition on projectable global types: parameterized operators should not include concrete limits – that is, we do not project types such as $\|_{i=1}^{3} a_i \xrightarrow{m} b_i$. This can be easily overcome by first expanding such applications to a non-parameterized form, and applying the projection function to the result. The previous example would be transformed into $a_1 \xrightarrow{m} b_1 \parallel a_2 \xrightarrow{m} b_2 \parallel a_3 \xrightarrow{m} b_3$ before being projected onto the participants.

8. Type Checking

So far, we have shown how to extract local types from Lang-A programs (Section 6.1) and also how to extract local types from global types using the projection function. By comparing the inferred and the projected local type, we can check a program's conformance to a given global type. To perform the comparison, we reduce the local types to a normal form according to the rules given in Tables 8, 9 and 10. This normal form neither includes τ terms (which come from applying the projection function), nor unnecessary parentheses. The reduction rules also replace complex operator applications with simpler ones, and bring the terms of commutative operators to a deterministic order.

Theorem 1 (Weak Normalization)**.** For any local type $L \in \mathscr{L}$ in System-A, there exists a finite sequence of reduction steps which brings the type to a normal form.

Rules for eliminating τ and getting rid of superfluous parentheses are shown in Tables 8 and 9, respectively. More complicated cases are those

Table 9: *Rules for eliminating parentheses from local types.*

$(L_1\ op \cdots op\ L_{n-1})\ op'\ L_n$	$\longrightarrow_{norm}$	$L_1\ op \cdots op\ L_{n-1}\ op'\ L_n$			
$L_1\ op'\ (L_2\ op \cdots op\ L_n)$	$\longrightarrow_{norm}$	$L_1\ op'\ L_2\ op \cdots op\ L_n$			
		where $op, op' \in \{\|, \otimes, \oplus, ;\}$ and $op' <_p op$ with $\| <_p \oplus <_p \otimes <_p ;$			
$\big((L)\big)$	$\longrightarrow_{norm}$	(L)			
$\big(\overset{n}{\underset{i=1}{OP}} L\big)$	$\longrightarrow_{norm}$	$\overset{n}{\underset{i=1}{OP}}(L)$	where $OP \in \{\|, \oplus, \otimes, \circledcirc\}$		
(L^n)	$\longrightarrow_{norm}$	$(L)^n$	(L^*)	$\longrightarrow_{norm}$	$(L)^*$
$(a!m)$	$\longrightarrow_{norm}$	$a!m$	$(a?m)$	$\longrightarrow_{norm}$	$a?m$

where types are structurally different, yet semantically equivalent. With $\equiv$ denoting trace equivalence (explained later), it is clear that both sides of

$$(L_1\ \oplus\ L_2)\ ;\ (L_3\ \oplus\ L_4) \equiv L_1\ ;\ L_3\ \oplus\ L_1\ ;\ L_4\ \oplus\ L_2\ ;\ L_3\ \oplus\ L_2\ ;\ L_4$$

describe the same behavior. Rule (R-Distributive) in Table 10 ensures that such equivalences do not enable infinite reduction sequences, by effectively disallowing one of the two directions. Additionally, rule (R-Simple-Paral) comes from the fact that the concurrent execution of two message sends/receives is effectively the same as shuffling them. The obvious extensions hold for sequences of operator applications (but not the parameterized versions).

A more complex case of semantic equivalence is introduced by the commutativity of operators $\oplus$, $\otimes$, and $\|$. This is because we need to avoid reduction steps of the form $L_1\ op\ L_2 \longrightarrow_{norm} L_2\ op\ L_1$ with $op \in \{\oplus, \otimes, \|\}$ as that would allow for infinite reduction sequences. By establishing an ordering among local types, we can reduce equivalent applications of an operator to a form where arguments always appear in a specific order. This is dealt with by rule (R-Commute), which uses the total order implied by Lemma 1.

Lemma 1 (Coding)**.** For each type L in our language of local types $\mathscr{L}$, there exists a unique code $c(L) \in \mathbb{N}$.

We give the proof of Lemma 1 in Appendix A. Using the reduction rules mentioned above, we can now prove the weak normalization property for local types in System-A.

Table 10: *Reductions of local types due to semantic equivalence.*

$L_1 \oplus L_2$	$\longrightarrow_{norm}$	L_1	where $L_1 = L_2$
$\overset{c_2}{\underset{i=c_1}{\circledcirc}} L$	$\longrightarrow_{norm}$	L^x	if i not free in L with $x = c_2 - c_1 + 1$ and c_1, c_2 known constants
$\overset{n}{\underset{i=1}{\circledcirc}} L$	$\longrightarrow_{norm}$	L^n	if i not free in L and n a parameter
$L_1 \otimes L_2$	$\longrightarrow_{norm}$	$(L_1 \,;\, L_2) \oplus (L_2 \,;\, L_1)$	(R-Distributive)
$(L_1 \oplus L_2) \,;\, L_3$	$\longrightarrow_{norm}$	$(L_1 \,;\, L_3) \oplus (L_2 \,;\, L_3)$	
$(L_1 \oplus L_2) \parallel L_3$	$\longrightarrow_{norm}$	$(L_1 \parallel L_3) \oplus (L_2 \parallel L_3)$	
$p_1\, op_1\, m_1 \parallel p_2\, op_2\, m_2$	$\longrightarrow_{norm}$	$p_1\, op_1\, m_1 \otimes p_2\, op_2\, m_2$	(R-Simple-Paral) where $op_1, op_2 \in \{!, ?\}$, p_1, p_2 are participating actors, and $m_1, m_2 \in$ message sorts

$$\frac{\exists i \text{ s.t. } c(L_i) > c(L_{i+1}) \qquad op \in \{\oplus, \otimes, \parallel\}}{L_1\, op, \ldots, op\, L_k \longrightarrow_{norm} L_{i_1}\, op, \ldots, op\, L_{i_k} \text{ s.t. } c(L_{i_1}) < \cdots < c(L_{i_k})} \quad \text{(R-Commute)}$$

Proof. (Weak Normalization)

Procedure NORMALIZE in Algorithm 1 brings the given local type to a normal form. In lines 2-6, it repeatedly applies operator precedence rules to eliminate parentheses (Table 9), applies the distributive law and other operator expansion laws (Table 10) as well as τ-eliminations (Table 8). We can show that the loop in lines 2-6 terminates, by observing that

(i) types are finite in length;

(ii) except for (R-Distributive), (R-Simple-Paral) and (R-Commute), all rules strictly shorten the type;

(iii) rule (R-Commute) is not applied in the loop, and

(iv) rules (R-Distributive) and (R-Simple-Paral) systematically reduce some cases of shuffling and concurrency to choices, and hence cannot produce repetitions.

After the loop, procedure ORDER handles operator commutativity by reordering the operator arguments (line 24) according to the coding given in Lemma 1. The following observations imply that ORDER terminates and that the produced type cannot be reduced any further:

1. line 11 is the base case (termination);
2. lines 14, 16, 18 and 21 get closer to the base case;
3. line 24 is only executed after the L_is have been normalized (line 21);
4. lines 2–6 converge and the loop terminates;
5. lines 2–6 need not be repeated as a result of ORDER. □

Corollary 1 (Strong Normalization)**.** Every local type $L \in \mathscr{L}$ has a unique normal form L_{norm}.

Proof. The normalization process given in Algorithm 1 is deterministic, and its output depends solely on the input type L. Hence the output is unique to the input type. □

It remains to show that a type and its normal form describe the same behavior. In order to do this, we need to define the semantics of local types, and show that the normalization process does not semantically alter a type.

On the semantics of local types. Recall that an *environment* $\Delta_P = \{a{:}L_a, \dots\}$ maps every actor in the program P to its local type. Also, observe that there is a clear correspondence between the constructs of Lang-A and those of local types (Table 5). This means we can define the trace generation relation $\rightsquigarrow$ on environments by essentially following the reduction rules of Lang-A (Table 4). For an environment Δ_P as above, this idea gives a reduction relation such that the code of the actors is simultaneously reduced with the respective types. The only extra requirement is that when an actor takes a non-observable action (assignment, expression evaluation etc.), the respective type takes a silent step. Restricting this reduction relation only to the local types of an environment, we get the operational semantics of local types; these are formally given in Appendix B.

Algorithm 1 Normalization of local types.

Input: A local type L.
Output: The respective unique normal form L_{norm}.

procedure NORMALIZE(L)
  repeat
    erase extraneous parentheses according to Table 9
    apply the τ-eliminations of Table 8
    apply the rules of Table 10 except (R-Commute)
  until no further change is possible in L
  ORDER(L)
end procedure

procedure ORDER(L)
  if $L = a!m$ or $L = a?m$ **then**
    return;
  else if $L = \overset{n}{\underset{i=s}{OP}} L_i$ where $OP \in \{\circledcirc\ ,\ \oplus\ ,\ \|\ ,\ \otimes\}$ **then**
    ORDER(L_i)
  else if $L = L_1^n$ or $L = L_1^*$ **then**
    ORDER(L_1)
  else if $L = (L')$ **then**
    ORDER(L')
  else if $L = L_1\ op \ldots op\ L_k$ where $op \in \{;\ ,\ \oplus\ ,\ \|\}$ **then**
    for $i = 1 \ldots k$ **do**
      ORDER(L_i)
    end for
    if $op \neq\ ;$ **then**
      reorder L according to (R-Commute)
    end if
  end if
end procedure

Theorem 2 (Type Normalization Preserves Semantics)**.** A type L and its normal form L_{norm} are trace equivalent.

Informally, two local types are *trace equivalent* [20, 10, 1] when they can be used interchangeably, in the sense that when placed within any given context, they both produce the same set of traces. Although we omit a formal proof for the sake of brevity, Theorem 2 can be shown by induction

on the length of the normalization process. Following the semantics in the appendix, one can verify that each normalization step results in an equivalent type (i.e. producing the same set of traces).

The similarity between the semantics of local types and those of Lang-A furthermore suggests the following theorem:

Theorem 3 (Subject Reduction)**.** Fix a program P, such that $P : \Delta$. Then, if $P \rightsquigarrow^* P'$ as per the language semantics (Table 4) with $P' : \Delta'$, we have that $\Delta \rightsquigarrow^* \Delta'$.

Since $\rightsquigarrow^*$ is the reflexive transitive closure of $\rightsquigarrow$, the above can be shown by induction on the number of reduction steps the program takes. Sketching the proof: because of the way the semantics of local types are formed, we can see that each possible reduction step of the program (Table 4) can be juxtaposed to a reduction step of the respective local type (Appendix B).

9. Global Type Realization

In this section, we discuss the properties that a global type must satisfy in order to be *projectable*. Applying the projection function to a *projectable* global type will result in local types for the participants whose combined behavior is consistent with the global type—a fact we show in Section 10.

Before we proceed, it is important to observe that the side conditions in the rules of Table 6 ensure well-typed programs, whereas the conditions we describe here ensure well-formed global types. The subsequent discussion of projectability criteria uses the following notions:

- We extend the projection function onto events and write $e \triangleright p$ to denote the projection of event e onto the participant p using rule (P-Interaction).
- Since a trace is simply a sequence of events of the form $p \xrightarrow{m} q$, we extend the projection function onto traces in the natural way. We write $t \triangleright p$ to denote the projection of trace t onto p using rules (P-Seq) and (P-Interaction). When $t \triangleright p$ contains only τ terms, we shorten the description by writing $t \triangleright p = \tau$ (empty projection).
- For a trace t, let $\mathit{first}\,(t)$ and $\mathit{last}\,(t)$ denote the first and last event of t respectively. Note that as discussed in Section 4.2, we only deal with finite traces.

- Recall that the set of traces a global type G can produce is denoted by $tr\,(G)$. Abusing notation, the set of events that appear first in traces of G is denoted $\mathit{first}\,(G) = \{\mathit{first}\,(t) \mid t \in tr\,(G)\}$. Similarly, the set of events that appear last in traces of G is denoted $\mathit{last}\,(G)$.

9.1. Sequentiality Criterion

The purpose of this criterion is to ensure that the sequential constructs of a global type retain their sequential semantics after projection. As an example problematic case, consider $G_1 = a \xrightarrow{m_1} b \;;\; c \xrightarrow{m_2} d$. Without the use of some covert coordination channel (for example by implementing a barrier mechanism), it is impossible for c to know when b has received the message. The two events $a \xrightarrow{m_1} b$ and $c \xrightarrow{m_2} d$ are impossible to order using our projection function, as the resulting environment would be $\Delta_1 = \{a : b!m_1,\, b : a?m_1,\, c : d!m_2,\, d : c?m_2\}$, which allows c to send m_2 to d before m_1 is received at b. G_1 does not satisfy the sequentiality criterion and thus is not projectable.

However, the case $G_2 = a \xrightarrow{m_1} b \;;\; a \xrightarrow{m_2} b$ is not problematic, since b can sequence the order of receiving messages. The following definition captures the conditions under which events are guaranteed to respect the sequencing restrictions imposed in a global type, when the latter is projected onto individual actors.

Definition 5 (Sequentially Projectable Global Type)**.** The set $P^{;}$ of *sequentially projectable* global types is defined inductively as follows:

$$
\begin{cases}
p_1 \xrightarrow{m} p_2 \in P^{;} \quad \forall p_1, p_2 \in \Pi \\
p_1 \xrightarrow{m_1} p_2 \;;\; p_2 \xrightarrow{m_2} p_3 \in P^{;} \quad \forall p_1, p_2, p_3 \in \Pi \\
p_1 \xrightarrow{m_1} p_2 \;;\; p_3 \xrightarrow{m_2} p_2 \in P^{;} \quad \forall p_1, p_2, p_3 \in \Pi \\
\\
G_1 \;;\; G_2 \in P^{;} \quad \mathit{iff} \quad \left(e_1 \;;\; e_2 \in P^{;} \quad \forall e_1 \in \mathit{last}\,(G_1)\,,\; e_2 \in \mathit{first}\,(G_2)\right) \\
\overset{n_2}{\underset{i=n_1}{\circledcirc}}\, G_i \in P^{;} \quad \mathit{iff} \quad \left(e_1 \;;\; e_2 \in P^{;} \quad \forall e_1 \in \mathit{last}\,(G_i[1/i])\,,\; e_2 \in \mathit{first}\,(G_i[2/i])\right) \\
G^n \in P^{;} \quad \mathit{iff} \quad \left(e_1 \;;\; e_2 \in P^{;} \quad \forall e_1 \in \mathit{last}\,(G)\,,\; e_2 \in \mathit{first}\,(G)\right) \\
G^* \in P^{;} \quad \mathit{iff} \quad \left(e_1 \;;\; e_2 \in P^{;} \quad \forall e_1 \in \mathit{last}\,(G)\,,\; e_2 \in \mathit{first}\,(G)\right)
\end{cases}
$$

where Π denotes the set of participating actors.

We illustrate the third case of the definition above with a type that is in $P^{;}$. Consider the global type $(a \xrightarrow{m} b \parallel c \xrightarrow{m} b) \;;\; (b \xrightarrow{m} l \parallel b \xrightarrow{m} k)$.

It is easy to see that $\mathit{last}(a \xrightarrow{m} b \parallel c \xrightarrow{m} b) = \{a \xrightarrow{m} b,\ c \xrightarrow{m} b\}$ and $\mathit{first}(b \xrightarrow{m} l \parallel b \xrightarrow{m} k) = \{b \xrightarrow{m} l,\ b \xrightarrow{m} k\}$, so that all four sequences (e.g., $a \xrightarrow{m} b\ ;\ b \xrightarrow{m} k$) are in $P^{;}$ according to the first two lines of Definition 5.

9.2. Choice Criterion

The purpose of this criterion is to ensure that projecting $G_1 \ \oplus \ G_2$ maintains the choice semantics, meaning that all participants can recognize which branch of the choice operator they need to take during execution. As an example of a type that does not satisfy this criterion, consider

$$G = (a \xrightarrow{m_1} b\ ;\ b \xrightarrow{k} c\ ;\ c \xrightarrow{t_1} d) \ \oplus \ (a \xrightarrow{m_2} b\ ;\ b \xrightarrow{k} c\ ;\ c \xrightarrow{t_2} d).$$

Here, a and b know which branch they are on, because on the left branch b receives a message of type m_1 from a, while on the branch on the right it receives a message of type m_2. However, from that point on, b behaves identically with respect to c, which has no way of telling whether the message to send to d should be of type t_1 or t_2. We call the first point at which two traces differ with respect to a given participant the *distinctive point*, which can be undefined if no such point exists.

Before giving the formal definition of the distinctive point, we first introduce some auxiliary notation. Let $t = e_1; \ldots ; e_n$ denote a trace and p a participant; then we have the projection $t \ \triangleright \ p = e_1 \triangleright p; \ldots ; e_n \triangleright p$. For any of the individual event projections, $\mathit{src}(\cdot)$ returns the index of the event in the original trace where the projection came from. In other words, for the projected events comprising $t \ \triangleright \ p$, it is $\mathit{src}(e_i \ \triangleright \ p) = i$ with $1 \leq i \leq n$. We also write $t \ \triangleright^{\tau} \ p$ to denote the result of removing τ from $t \ \triangleright \ p$. Essentially, $t \ \triangleright^{\tau} \ p$ is the result of first projecting the trace, then applying the rules of Table 8 to the result. In this case, $\mathit{src}(\cdot)$ is useful to recover the indices as they were before eliminating τ.

Definition 6 (Distinctive Point)**.** Let t_1 and t_2 be traces. For a participant p, take the τ-free projections $t_1 \ \triangleright^{\tau} \ p = s_1; \ldots ; s_k$ and $t_2 \ \triangleright^{\tau} \ p = u_1; \ldots ; u_l$. Then the *distinctive point* for the participant p with respect to the pair of traces t_1, t_2 is defined as

$$d_{t_1,t_2}(p) = (i, j) \text{ with } i = \mathit{src}(s_q),\ j = \mathit{src}(u_q) \quad \text{for the unique } (s_q,\ u_q) \text{ s.t. } q = \min\{z \mid s_z \neq u_z\}.$$

In the case where $t_1 \ \triangleright \ p = \tau$, or $t_2 \ \triangleright \ p = \tau$, or $t_1 \ \triangleright \ p = t_2 \ \triangleright \ p$, no such (i, j) exists and the distinctive point is undefined.

The definition that follows captures the conditions under which the choice semantics are maintained after projection. The first bullet deals with the non-parameterized version of the choice operator $\oplus$. Item (i) captures the case where a participant p is the first one acting on the two branches, in which case it must inform the others of the branch they are on. It does so by either sending a different message, or by sending to a different actor in each case. Note that the same actor must inform the others on both branches. Item (ii) captures the case where p is not the first one to act, in which case it must be informed of the branch it is on and the distinctive point should be a suitable receive event.

The second bullet inductively uses the first one to define choice-wise projectability in the parameterized case of choice $\oplus$.

Definition 7 (Choice-Wise Projectable Global Type)**.** The set $P^{\oplus}$ of *choice-wise projectable* global types is defined inductively as follows:

- $G = G_1 \ \oplus \ G_2 \in P^{\oplus}$ *iff* $\forall p \in \Pi$, either of the following is true:

  (i) $\forall e_1 \in \mathit{first}\,(G_1)\,,\ e_2 \in \mathit{first}\,(G_2)\ :\ e_1 = p \xrightarrow{m} q,\ e_2 = p \xrightarrow{m'} q'$
  where $p \neq q$ and $p \neq q'$and $(q \neq q'$ or $m \neq m')$

  (ii) $\forall t_1 = (s_1, \ldots, s_{k_1}) \in tr\,(G_1)\,,\ t_2 = (u_1, \ldots, u_{k_2}) \in tr\,(G_2)\,,$
  either $d_{t_1,t_2}(p)$ is undefined, or
  $d_{t_1,t_2}(p) = (i, j)$ and $s_i = q \xrightarrow{m} p,\ u_j = q' \xrightarrow{m'} p$
  where $s_i \in tr\,(G_1)\,,\ u_j \in tr\,(G_2)$
  and $p \neq q$ and $p \neq q'$and $(q \neq q'$ or $m \neq m')$

- $G = \bigoplus_{i=n_1}^{n_2} G_i \in P^{\oplus} \quad$ *iff* $\quad \big(G_i[1/i] \ \oplus \ G_i[2/i]\big) \in P^{\oplus}$

where Π is the set of participating actors.

9.3. Shuffle Criterion

To be able to project the shuffle operators while maintaining the semantics, we rely on both the choice and the sequencing criteria. Recall that we define the shuffle operator as

$$G_1 \ \otimes \ G_2 = (G_1 \,;\, G_2) \ \oplus \ (G_2 \,;\, G_1).$$

For this to be projectable, it has to be the case that G_1 and G_2 can be sequenced both ways, and also that the right hand side satisfies the choice projectability criteria.

Definition 8 (Shuffle-Wise Projectable Global Type)**.** The set $P^{\otimes}$ of *shuffle-wise projectable* global types is defined inductively as follows:

- $G_1 \otimes G_2 \in P^{\otimes}$ *iff* all of the following are true:
  1. $G_1 \,;\, G_2 \in P^{;}$
  2. $G_2 \,;\, G_1 \in P^{;}$
  3. $G_1 \oplus G_2 \in P^{\oplus}$
- $\bigotimes_{i=n_1}^{n_2} G_i \in P^{\otimes}$ *iff* $G_i[1/i] \otimes G_i[2/i] \in P^{\otimes}$.

Notice how item (3) above works: if an actor can tell whether it is on G_1 or G_2, then it is also able to tell the order in which G_1 and G_2 appear.

9.4. Concurrent Composability Criterion

As an example of what can go wrong when composing two global types with the $\|$ operator, consider the following example:

$$G = \big((a \xrightarrow{m_1} b \,;\, b \xrightarrow{k_1} c) \;\oplus\; (a \xrightarrow{m_2} b \,;\, b \xrightarrow{k_2} c)\big) \;\|\; a \xrightarrow{m_1} b.$$

The intended behavior of G is that a chooses whether to send a message of type m_1 or m_2 to b, which in turn decides whether to send c a message of type k_1 or k_2. Concurrently with this, an additional m_1 is sent from a to b. Assume that as far as $\oplus$ is concerned, a decides to send m_2 to b. It is then obvious how the additional concurrent event $a \xrightarrow{m_1} b$ might confuse b to simultaneously take both branches of the choice operator.

In general, the problem appears when actions in one concurrent branch affect choices made on another. Global types that do not exhibit this problem are *concurrently projectable*.

Definition 9 (Concurrently Projectable Global Types)**.** For two global types G_1 and G_2, we have that $(G_1 \parallel G_2)$ is *concurrently projectable* if there is no overlap between the distinctive points in G_1 and events in G_2. Formally, the set $P^{\parallel}$ of concurrently projectable global types is defined as follows:

- $G_1 \parallel G_2 \in P^{\parallel}$ *iff* $\forall t_1 = (e_1, \ldots, e_k),\ t_1' = (e_1', \ldots, e_{k'}') \in tr\,(G_1)\,,$
  $t_2 \in tr\,(G_2)\,,\ p \in \Pi,$ one of the following is true:

  (a) $d_{t_1,t_1'}(p)$ is undefined
  (b) $d_{t_1,t_1'}(p) = (i,j)$ and e_i, e_j' both have p as the sender
  (c) $d_{t_1,t_1'}(p) = (i,j)$ and $e_i \notin t_2$ and $e_j' \notin t_2$

- $\prod_{i=n_1}^{n_2}{}^{\parallel} G_i \in P^{\parallel}$ *iff* $\big(G_i[1/i] \parallel G_i[2/i]\big) \in P^{\parallel}$

where Π denotes the set of participating actors. Notice how this definition also deals with the concurrent composability of two Kleene starred types (the Kleene star entails a choice pertaining to loop entrance and exit).

9.5. Kleene Star Criterion

Use of the Kleene star in global types can result in protocols whose projection is unsafe, that is, can result in execution traces that are not part of the original global type. To avoid this, a global type must be such that the entry and exit conditions to the starred type can be identified by all participants. Determining whether this is the case requires inspection of not only the starred type itself, but also of what comes after the starred section. Observe that $G^*\,;\,G' \equiv G\,;\,(G^*\,;\,G')\ \oplus\ G'$, which means that both sequencing and choice constraints must be satisfied simultaneously.

Definition 10 (Kleene Star Projectable Global Types)**.** The set P^* of *Kleene star projectable* global types is defined to be the set of all global types of the form $G^*\,;\,G'$ for which the following are all true:

- $G^* \in P^{;}$
- $G\ \oplus\ G' \in P^{\oplus}$
- $G\,;\,G' \in P^{;}$

Recall that $P^{;}$ and $P^{\oplus}$ are the sets of sequentially and respectively choice-wise projectable types. As an example of a type that is not in P^{*}, consider $G = (a \xrightarrow{m} b \,;\, b \xrightarrow{m'} c)^{*} \,;\, c \xrightarrow{m''} d$ where c has no way of knowing whether it should wait for m' from b, or proceed immediately with sending the message m'' to d.

10. Correctness

The conditions discussed in the previous section are sufficient to ensure that the projection function generates local types which are functionally consistent with the global type. We call a global type that satisfies all of the above criteria *projectable*:

Definition 11 (Projectable Global Type)**.** The set PR of *projectable* global types is inductively defined in Table 11.

Table 11: *Inductive definition of the set PR of projectable global types. The sets $P^{;}$ of sequentially projectable, $P^{\oplus}$ of choice-wise projectable, $P^{\otimes}$ of shuffle-wise projectable, $P^{\|}$ of concurrently projectable, and P^{*} of Kleene star projectable global types $G \in \mathscr{G}$ are defined in Section 9. See Table 2 for the domains of the auxiliary symbols.*

$a \xrightarrow{m} b \in PR$		
$G^{n} \in PR$	*iff*	$G \in PR$ and $G^{n} \in P^{;}$
$(G_1 \,;\, G_2) \in PR$	*iff*	$G_1, G_2 \in PR$ and $(G_1 \,;\, G_2) \in P^{;}$
$(G_1 \,\oplus\, G_2) \in PR$	*iff*	$G_1, G_2 \in PR$ and $(G_1 \,\oplus\, G_2) \in P^{\oplus}$
$(G_2 \,\otimes\, G_2) \in PR$	*iff*	$G_1, G_2 \in PR$ and $(G_1 \,\otimes\, G_2) \in P^{\otimes}$
$(G_1 \parallel G_2) \in PR$	*iff*	$G_1, G_2 \in PR$ and $(G_1 \parallel G_2) \in P^{\|}$
$(G_1^{*} \,;\, G_2) \in PR$	*iff*	$G_1, G_2 \in PR$ and $(G_1^{*} \,;\, G_2) \in P^{*}$
$\overset{n_2}{\underset{i=n_1}{\circledcirc}} G_i \in PR$	*iff*	$G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \,;\, G_i[2/i]) \in P^{;}$
$\overset{n_2}{\underset{i=n_1}{\bigoplus}} G_i \in PR$	*iff*	$G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \,\oplus\, G_i[2/i]) \in P^{\oplus}$
$\overset{n_2}{\underset{i=n_1}{\bigotimes}} G_i \in PR$	*iff*	$G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \,\otimes\, G_i[2/i]) \in P^{\otimes}$
$\overset{n_2}{\underset{i=n_1}{\|}} G_i \in PR$	*iff*	$G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \parallel G_i[2/i]) \in P^{\|}$

We denote the environment resulting from the projection of G onto each one of the participants by Δ_G. That is, $\Delta_G = \{p : G \triangleright p\}_{p \in \Pi}$ where Π is the set of participating actors.

Definition 12 (Environment Traces)**.** The set of traces $tr(\Delta)$ producible by an environment Δ is the union of the sets of traces producible by the local types in Δ, as per the semantics found in Appendix B.

Theorem 4 formalizes our intuition that under the constraints of Table 11, the projection function of Table 7 is correct; that is, the projected environment produces the same set of traces as the global type.

Theorem 4. $G \in PR \Rightarrow tr(G) = tr(\Delta_G)$

Proof. We prove Theorem 4 by induction on the structure of global types. The base case consists of two parts:

- $G = a \xrightarrow{m} b$ with $a \neq b$, projecting onto the environment $\Delta_G = \{a : b!m,\ b : a?m\}$. Obviously, $tr(G) = tr(\Delta_G) = \{a \xrightarrow{m} b\}$.
- $G = a \xrightarrow{m} a$, projecting onto $\Delta_G = \{a : a!m\,;\, a?m\}$ and again, $tr(G) = tr(\Delta_G) = \{a \xrightarrow{m} a\}$.

We now treat each of the inductive steps separately. For $G \in PR$, the inductive hypothesis is that the theorem holds for each of the sub-components $G_i \in PR$ comprising G.

[Seq] $\quad (G_1\,;\,G_2) \in PR \Rightarrow tr(G_1\,;\,G_2) = tr(\Delta_{G_1;G_2})$

Let $t \in tr(G_1\,;\,G_2)$

$\Leftrightarrow\ \exists t_1 \in tr(G_1), t_2 \in tr(G_2)$ s.t. $t = t_1\,;\,t_2$ (def. of sequencing)

$\Leftrightarrow\ t_1 \in tr(\Delta_{G_1})$ and $t_2 \in tr(\Delta_{G_2})$ (inductive hypothesis)

Observe that

$$\Delta_{G_1;G_2} = \begin{cases} \dots \\ a : (G_1 \triangleright a)\ ;\ (G_2 \triangleright a), \\ b : (G_1 \triangleright b)\ ;\ (G_2 \triangleright b), \\ c : \underbrace{(G_1 \triangleright c)}_{\Delta_{G_1}}\ ;\ \underbrace{(G_2 \triangleright c)}_{\Delta_{G_2}} \end{cases}$$

and that the part on the left of the semi-colons is Δ_{G_1}, while the part on the right is Δ_{G_2}, which means that they produce t_1 and t_2 respectively. Now since $G_1; G_2 \in PR$, we have that $G_1; G_2 \in P^{;}$ which means that $last\,(t_1) = a \xrightarrow{m_1} b$ and $first\,(t_2) \in \{b \xrightarrow{m_2} c\,,\, c \xrightarrow{m_2} b\}$. Hence, for the environment above, we have: having produced t_1 at the semi-colon, the nature of the connective events ($last\,(t_1)$ and $first\,(t_2)$) forces the execution of $\Delta_{G_1;G_2}$ to produce exactly the trace $t_1\,;\,t_2$. We complete the proof by observing that this argument holds for any t_1, t_2 as above, because of the definition of $P^{;}$.

[Exp] $\quad G^n \in PR \Rightarrow tr\,(G^n) = tr\,(\Delta_{G^n})$

From the definition of PR, we have $G \in PR$ and $G^n \in P^{;}$. Notice how the latter implies that $(G\,;\,\ldots\,;\,G) \in P^{;}$, thus the proof is similar to case [Seq] above.

[Choice] $\quad (G_1 \,\oplus\, G_2) \in PR \Rightarrow tr\,(G_1 \,\oplus\, G_2) = tr\,(\Delta_{G_1 \oplus G_2})$.

From the definition of PR, we have $G_1, G_2 \in PR$ and $(G_1 \,\oplus\, G_2) \in P^{\oplus}$.

$(\subseteq)$
Let $t \in tr\,(G_1 \,\oplus\, G_2)$. By definition of the choice operator, $t \in tr\,(G_1)$ or $t \in tr\,(G_2)$. It is easy to see that $t \in \Delta_{G_1 \oplus G_2}$, by noticing (i) the latter equals $\{p : (G_1 \triangleright p) \,\oplus\, (G_2 \triangleright p)\}_{p \in \Pi}$; (ii) the part on the left of the choice operator is Δ_{G_1}, while the part on the right is Δ_{G_2}; (iii) by the inductive hypothesis, $tr\,(\Delta_{G_1}) = tr\,(G_1)$ and $tr\,(\Delta_{G_2}) = tr\,(G_2)$.

$(\supseteq)$
Let $t \in tr\,(\Delta_{G_1 \oplus G_2})$. By definition of the $\oplus$ operator, we have $t \in tr\,(\Delta_{G_1})$ or $t \in tr\,(\Delta_{G_2})$, and by the inductive hypothesis, $tr\,(G_1) = tr\,(\Delta_{G_1})$ and $tr\,(G_2) = tr\,(\Delta_{G_2})$. Pick any two participants p, q in the system and let $t_1 \in tr\,(G_1)$ and $t_2 \in tr\,(G_2)$. Since $G_1 \oplus G_2 \in P^{\oplus}$, we see the following cases:

- $first\,(t_1) = p \xrightarrow{m} q$ and $first\,(t_2) = p \xrightarrow{m'} q'$. These two events are different from the perspectives of p and q alike. Hence, either they are both executing t_1, or they are both executing t_2.
- $d_{t_1,t_2}(p)$ is undefined. Then two cases: either $t_1 \triangleright p = t_2 \triangleright p$ so it doesn't matter which one p is executing; or one of the projections is empty (τ) but the other isn't, so p definitely knows which branch it is on.

- $d_{t_1,t_2}(p) = (i,j)$ with $s_i = q \xrightarrow{m} p \ \in t_1$, $u_j = q' \xrightarrow{m'} p \ \in t_2$. These events are different from the perspective of p, as they are from the perspective of q, so they are both on t_1, or (both) on t_2.

Notice that the roles of p, q above can be interchanged. Since the cases above are true for all pairs of participants, it is either $t = t_1$, or $t = t_2$ for some $t_1 \in tr\,(G_1)$, $t_2 \in tr\,(G_2)$ as above. By definition of the $\oplus$ operator, $t \in tr\,(G_1 \ \oplus \ G_2)$.

[Shuffle] $\quad (G_1 \ \otimes \ G_2) \in PR \Rightarrow tr\,(G_1 \ \otimes \ G_2) = tr\,(\Delta_{G_1 \otimes G_2})$

The definition of PR gives us $G_1, G_2 \in PR$ and $(G_1 \ \otimes \ G_2) \in P^{\otimes}$. Because shuffling can be expressed in terms of sequencing and choice, the proof falls back onto cases [Seq] and [Choice].

[Paral] $\quad (G_1 \parallel G_2) \in PR \Rightarrow tr\,(G_1 \parallel G_2) = tr\left(\Delta_{G_1 \parallel G_2}\right)$

From the definition of PR, we have $G_1, G_2 \in PR$ and $(G_1 \parallel G_2) \in P^{\parallel}$.

$(\subseteq)$

Let $t \in tr\,(G_1 \parallel G_2)$, which (by definition of the $\parallel$ operator) is an interleaving of some $t_1 \in tr\,(G_1)$ and some $t_2 \in tr\,(G_2)$. By the induction hypothesis, $t_1 \in tr\,(\Delta_{G_1})$ and $t_2 \in tr\,(\Delta_{G_2})$. Obviously $t \in tr\,(\{p : (G_1 \triangleright p \parallel G_2 \triangleright p)\}_{p \in \Pi})$ as the parts on the left of the $\parallel$ operator are Δ_{G_1} and those on the right are Δ_{G_2}.

$(\supseteq)$

Observe that $\Delta_{G_1 \parallel G_2} = \{p : (G_1 \triangleright p \parallel G_2 \triangleright p)\}_{p \in \Pi}$. Clearly the left side of the $\parallel$ operator corresponds to Δ_{G_1}, while the right corresponds to Δ_{G_2}. Because of this form, any trace $t \in tr\left(\Delta_{G_1 \parallel G_2}\right)$ will be a function of some $t_1, t_1' \in tr\,(\Delta_{G_1})$, $t_2 \in tr\,(\Delta_{G_2})$. By the inductive hypothesis, we have $t_1, t_1' \in tr\,(G_1)$ and $t_2 \in tr\,(G_2)$.

Assume that $t_1 = (e_1, \ldots, e_k)$ and $t_1' = (e_1', \ldots, e_{k'}')$. Because $(G_1 \parallel G_2) \in P^{\parallel}$, we have that for every participant p, one of the following is true:

- $d_{t_1,t_1'}(p)$ is undefined so t_2 cannot "confuse" p regarding choices concerning t_1 and t_1' (p knows if it is executing t_1 or t_1').
- $d_{t_1,t_1'}(p) = (i,j)$ and both e_1 and e_j' have p as the sender, so t_2 cannot "confuse" p regarding choices in t_1 and t_1'.
- $d_{t_1,t_1'}(p) = (i,j)$ and $e_i \notin t_2$ and $e_j' \notin t_2$ in which case nothing in t_2 can confuse p by definition.

Thus interleaving G_1 and G_2 does not alter choices made by p on traces produced by either, for any $p \in \Pi$. This means that $t_1 \parallel t_2 \in tr\,(G_1 \parallel G_2)$ – and symmetrically, $t'_1 \parallel t_2 \in tr\,(G_1 \parallel G_2)$[1]

[KleeneStar] $\quad (G_1^* \,;\, G_2) \in PR \Rightarrow tr\,(G_1^* \,;\, G_2) = tr\left(\Delta_{G_1^* \,;\, G_2}\right)$

Since the definition of PR implies both $G_1^* \in P^{;}$ and $G_1 \,;\, G_2 \in P^{;}$, sequencing as in $G_1 \,;\, \ldots \,;\, G_1 \,;\, G_2$ is dealt with by case [Seq] above. The choice part of the Kleene Star (see Section 9.5) is dealt with by case [Choice], since the definition of PR also implies that $G_1 \,\oplus\, G_2 \in P^{\oplus}$.

A note on global types that end with a Kleene Star, such that G_2 above is empty: This case can be reasoned about by extending the projection function in the natural way, so that the empty global type projects to the empty environment. We complete this part of the proof by observing that (a) both the empty type and the empty environment produce the empty set of traces, and (b) sequencing any type with the empty one trivially respects the sequencing constraints of Section 9.1.

[ParamSeq] $\quad G \in PR \Rightarrow tr\,(G) = tr\,(\Delta_G)$ where $G = \overset{n_2}{\underset{i=n_1}{\circledcirc}} G_i$.

$G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \,;\, G_i[2/i]) \in P^{;}$ hold from the definition of PR. Substituting 1 and 2 for i to generate G_1 and G_2 allows for the direct checking of the $P^{;}$ criteria, and an inductive argument lets the proof fall back onto case [Seq].

[ParamChoice] $\quad G \in PR \Rightarrow tr\,(G) = tr\,(\Delta_G)$ where $G = \overset{n_2}{\underset{i=n_1}{\bigoplus}} G_i$.

$G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \,\oplus\, G_i[2/i]) \in P^{\oplus}$ are true from the definition of PR. It is easy to see that substituting 1 and 2 for i to generate G_1, G_2 allows for the direct checking of the $P^{\oplus}$ criteria, and an inductive argument lets the proof fall back onto case [Choice].

[ParamShuffle] $\quad G \in PR \Rightarrow tr\,(G) = tr\,(\Delta_G)$ where $G = \overset{n_2}{\underset{i=n_1}{\bigotimes}} G_i$.

From the definition of PR, we have that $G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \,\otimes\, G_i[2/i]) \in P^{\otimes}$. It is easy to see that substituting 1 and 2

[1] we abuse notation, so that $t_1 \parallel t_2$ means "some interleaving of" traces t_1 and t_2.

for i to generate G_1, G_2 allows for direct checking of the $P^{\otimes}$ criteria, and an inductive argument has the proof fall back onto case [Shuffle].

[ParamParal] $\quad G \in PR \Rightarrow tr\,(G) = tr\,(\Delta_G)$ where $G = \mathop{\|}_{i=n_1}^{n_2} G_i$.

From the definition of PR, we have that $G_i[1/i], G_i[2/i] \in PR$ and $(G_i[1/i] \parallel G_i[2/i]) \in P^{\|}$. It is easy to see that substituting 1 and 2 for i allows for the direct checking of the $P^{\|}$ criteria, and an inductive argument lets the proof fall back onto case [Paral]. □

11. Conclusions and Future Work

We introduced System-A, a session-type system that allows for parameterized concurrency, where the number of participants, the types of messages sent, as well as the number of such messages are controlled by type parameters. We also demonstrated how to parameterize choice among various execution paths, so that the number and types of different paths to be taken may be unknown at compile time. Our system includes a novel shuffling operator, which expresses arbitrary reorderings of its arguments, again in a parameterized fashion. We furthermore described Lang-A, a programming language in which we can implement programs whose adherence to System-A specifications can be statically checked. A series of examples demonstrates that our approach allows us to specify and check more complicated interactions than prior work, such as the sliding window protocol and parallel resource locking/unlocking (Section 3).

In System-A, we can statically verify—without parameter instantiation—the compliance of implementations to protocols. We do this by first projecting (Section 7) the specification to parameterized types, and then comparing these projections against the types extracted from the program. We showed how to perform the latter step, i.e. type inference from Lang-A code (Section 6.1), and also demonstrated that structural equivalence of types in System-A is decidable. We presented this result in Section 8 by first showing weak, and subsequently strong normalization of local types. In Section 9 we discussed the conditions under which our projection function is correct, and proved their sufficiency in Section 10.

Future Work. Adding support for dynamic process creation is an important direction for future work. In its current form, System-A cannot express actor creation as a behavior, and global types assume that all participants already exist. Matching a created actor with its subsequent use in a type requires an extra step which is not obvious.

The semantic comparison of local types constitutes another practical consideration. Our normalization algorithm (Section 8) already includes many cases of semantically equivalent, yet structurally differing types (Table 10). Semantic comparison is unnecessary for the weak normalization proof, but would be useful in a practical setting where the user is interested in semantic adherence to a protocol. Specifically in the case where reordering of terms is possible as a result of operator commutativity, our suggested coding only serves as an existential proof. A more practical coding scheme could be developed, perhaps employing lexicographic ordering.

In its present form, System-A does not allow the arbitrary use of parameters. First of all, only indices that range over continuous integers are supported. Secondly, some all-to-all types of communication are not supported, due to the issues discussed in Section 7. These limitations can be overcome by allowing index sets, and making restrictions on them explicit. This would, for example, enable types with constraints such as

$$\|_{i \in I} \|_{j \in J} a_i \xrightarrow{m} a_j \quad I \cap J = \emptyset.$$

One can then reason about how these restrictions affect projection, which will produce different output depending on provided side-conditions such as "project onto a_k where $k \in I$".

Deniélou and Yoshida [21] propose a system where parameterization is achieved by means of quantification over *roles*. Roles are behavior specifications that are taken up by processes while they participate in a protocol, and processes are allowed to join and leave protocols dynamically. Their notation's expressiveness is limited when it comes to arbitrary, concurrency-induced interleavings of events. Nevertheless, incorporating their work into System-A would expand the applicability of the ideas presented here – even though towards a different direction.

Other possible extensions include support for session delegation and exception handling (in the sense of Carbone et al. [12]). It may also be possible to transfer the recent, precise realizability results [4] for choreographies [44]

to our parameterized specifications. Finally, other possible extensions concern the runtime monitoring application domain [18]. In particular, adding support for global assertions [8] can form the basis of a powerful theory for deriving local restrictions for each participant, which an asynchronous observer [19] can then enforce.

Acknowledgments

The authors would like to thank Karl Palmskog for his helpful comments. We would also like to thank the anonymous reviewers for their many suggestions, which have undoubtedly led to the maturement of this paper. This publication was made possible in part by sponsorships from the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084, and also the National Science Foundation under grant number CCF-1438982. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] G. Agha, I.A. Mason, S.F. Smith, C.L. Talcott, A foundation for actor computation, J. Funct. Program. 7 (1997) 1–72.

[2] F. Arbab, Reo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (2004) 329–366.

[3] H. Barendregt, S. Abramsky, D.M. Gabbay, T.S.E. Maibaum, H.P. Barendregt, Lambda calculi with types, in: Handbook of Logic in Computer Science, Oxford University Press, 1992, pp. 117–309.

[4] S. Basu, T. Bultan, M. Ouederni, Deciding choreography realizability, in: J. Field, M. Hicks (Eds.), POPL, ACM, 2012, pp. 191–202.

[5] A. Bejleri, Practical parameterised session types, in: J.S. Dong, H. Zhu (Eds.), ICFEM, volume 6447 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 270–286.

[6] A. Bejleri, Parameterised session types communication patterns: through the looking glass of session types, Ph.D. thesis, Imperial College London, 2012.

[7] L. Bettini, M. Coppo, L. D'Antoni, M.D. Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: F. van Breugel, M. Chechik (Eds.), CONCUR, volume 5201 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 418–433.

[8] L. Bocchi, K. Honda, E. Tuosto, N. Yoshida, A theory of design-by-contract for distributed multiparty interactions, in: P. Gastin, F. Laroussinie (Eds.), CONCUR, volume 6269 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 162–176.

[9] E. Bonelli, A.B. Compagnoni, Multipoint session types for a distributed calculus, in: G. Barthe, C. Fournet (Eds.), TGC, volume 4912 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 240–256.

[10] M. Boreale, R. De Nicola, R. Pugliese, Trace and testing equivalence on asynchronous processes, Inf. Comput. 172 (2002) 139–164. URL: `http://dx.doi.org/10.1006/inco.2001.3080`. doi:`10.1006/inco.2001.3080`.

[11] M. Carbone, K. Honda, N. Yoshida, Structured communication-centred programming for web services, in: R. De Nicola (Ed.), ESOP, volume 4421 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 2–17.

[12] M. Carbone, K. Honda, N. Yoshida, Structured interactional exceptions in session types, in: F. van Breugel, M. Chechik (Eds.), CONCUR, volume 5201 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 402–417.

[13] M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, in: R. Giacobazzi, R. Cousot (Eds.), POPL, ACM, 2013, pp. 263–274.

[14] M. Carbone, N. Yoshida, K. Honda, Asynchronous session types: Exceptions and multiparty interactions, in: M. Bernardo, L. Padovani, G. Zavattaro (Eds.), SFM, volume 5569 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 187–212.

[15] G. Castagna, M. Dezani-Ciancaglini, L. Padovani, On global types and multi-party sessions, in: R. Bruni, J. Dingel (Eds.), FMOODS/FORTE, volume 6722 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 1–28.

[16] G. Castagna, M. Dezani-Ciancaglini, L. Padovani, On global types and multi-party session, Logical Methods in Computer Science 8 (2012).

[17] M. Charalambides, P. Dinges, G. Agha, Parameterized concurrent multi-party session types, in: N. Kokash, A. Ravara (Eds.), FOCLASA, volume 91 of *EPTCS*, pp. 16–30.

[18] T.C. Chen, L. Bocchi, P.M. Deniélou, K. Honda, N. Yoshida, Asynchronous distributed monitoring for multiparty session enforcement, in: R. Bruni, V. Sassone (Eds.), TGC, volume 7173 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 25–45.

[19] T.C. Chen, K. Honda, Specifying stateful asynchronous properties for distributed programs, in: M. Koutny, I. Ulidowski (Eds.), CONCUR, volume 7454 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 209–224.

[20] R. De Nicola, M. Hennessy, Testing equivalences for processes, Theor. Comput. Sci. 34 (1984) 83–133. URL: `http://dx.doi.org/10.1016/0304-3975(84)90113-0`. doi:`10.1016/0304-3975(84)90113-0`.

[21] P.M. Deniélou, N. Yoshida, Dynamic multirole session types, in: T. Ball, M. Sagiv (Eds.), POPL, ACM, 2011, pp. 435–446.

[22] P.M. Deniélou, N. Yoshida, Multiparty session types meet communicating automata, in: H. Seidl (Ed.), ESOP, volume 7211 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 194–213.

[23] M. Dezani-Ciancaglini, E. Giachino, S. Drossopoulou, N. Yoshida, Bounded session types for object oriented languages, in: F.S. de Boer, M.M. Bonsangue, S. Graf, W.P. de Roever (Eds.), FMCO, volume 4709 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 207–245.

[24] P. Dinges, G. Agha, Scoped synchronization constraints for large scale actor systems, in: M. Sirjani (Ed.), COORDINATION, volume 7274 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 89–103.

[25] P. Eugster, T. Frischbier, S. Buchmann, Sound transformations for federated objects, in: M.B. Dwyer (Ed.), OOPSLA, ACM, 2012.

[26] N. Francez, Fairness, Texts and Monographs in Computer Science, Springer, 1986. URL: http://dx.doi.org/10.1007/978-1-4612-4886-6. doi:10.1007/978-1-4612-4886-6.

[27] S. Frølund, Coordinating distributed objects - an actor-based approach to synchronization, MIT Press, 1996.

[28] S. Frølund, G. Agha, A language framework for multi-object coordination, in: O. Nierstrasz (Ed.), ECOOP, volume 707 of *Lecture Notes in Computer Science*, Springer, 1993, pp. 346–360.

[29] S.J. Gay, M. Hole, Subtyping for session types in the pi calculus, Acta Inf. 42 (2005) 191–225.

[30] S.J. Gay, V.T. Vasconcelos, Linear type theory for asynchronous session types, J. Funct. Program. 20 (2010) 19–50.

[31] S.J. Gay, V.T. Vasconcelos, A. Ravara, N. Gesbert, A.Z. Caldeira, Modular session types for distributed object-oriented programming, in: M.V. Hermenegildo, J. Palsberg (Eds.), POPL, ACM, 2010, pp. 299–312.

[32] K. Gödel, Über eine bisher noch nicht benützte erweiterung des finiten standpunktes, Dialectica (1958) 280287.

[33] M. Hennessy, Semantics of programming languages - an elementary introduction using structural operational semantics, Wiley, 1990.

[34] K. Honda, Types for dyadic interaction, in: E. Best (Ed.), CONCUR, volume 715 of *Lecture Notes in Computer Science*, Springer, 1993, pp. 509–523.

[35] K. Honda, V.T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: C. Hankin (Ed.), ESOP, volume 1381 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 122–138.

[36] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: G.C. Necula, P. Wadler (Eds.), POPL, ACM, 2008, pp. 273–284.

[37] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, K. Honda, Type-safe eventful sessions in java, in: T. D'Hondt (Ed.), ECOOP, volume 6183 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 329–353.

[38] R. Hu, N. Yoshida, K. Honda, Session-based distributed programming in java, in: J. Vitek (Ed.), ECOOP, volume 5142 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 516–541.

[39] D. Kouzapas, N. Yoshida, K. Honda, On asynchronous session semantics, in: R. Bruni, J. Dingel (Eds.), FMOODS/FORTE, volume 6722 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 228–243.

[40] I. Lanese, C. Guidi, F. Montesi, G. Zavattaro, Bridging the gap between interaction- and process-oriented choreographies, in: A. Cerone, S. Gruner (Eds.), SEFM, IEEE Computer Society, 2008, pp. 323–332.

[41] W.C. Lynch, Computer systems: Reliable full-duplex file transmission over half-duplex telephone line, Commun. ACM 11 (1968) 407–410. URL: `http://doi.acm.org/10.1145/363347.363366`. doi:`10.1145/363347.363366`.

[42] G. Milicia, V. Sassone, Jeeg: temporal constraints for the synchronization of concurrent objects, Concurrency - Practice and Experience 17 (2005) 539–572.

[43] D. Mostrous, N. Yoshida, K. Honda, Global principal typing in partially commutative asynchronous sessions, in: G. Castagna (Ed.), ESOP, volume 5502 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 316–332.

[44] C. Peltz, Web services orchestration and choreography, IEEE Computer 36 (2003) 46–52.

[45] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60-61 (2004) 17–139.

[46] R.L. Probert, K. Saleh, Synthesis of communication protocols: Survey and assessment, IEEE Trans. Computers 40 (1991) 468–476.

[47] R. Pucella, J.A. Tov, Haskell session types with (almost) no class, in: A. Gill (Ed.), Haskell, ACM, 2008, pp. 25–36.

[48] F. Puntigam, Types for active objects based on trace semantics, in: E.N. et al. (Ed.), Proceedings of the Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96), IFIP WG 6.1, Chapman & Hall, Paris, France, 1996. URL: `http://www.complang.tuwien.ac.at/franz/papers/Punt96b.ps.gz`.

[49] F. Puntigam, Coordination requirements expressed in types for active objects, in: ECOOP, pp. 367–388.

[50] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: C. Halatsis, D.G. Maritsas, G. Philokyprou, S. Theodoridis (Eds.), PARLE, volume 817 of *Lecture Notes in Computer Science*, Springer, 1994, pp. 398–413.

[51] G. Tel, Introduction to Distributed Algorithms, Cambridge university press, 2000.

[52] V.T. Vasconcelos, S.J. Gay, A. Ravara, Type checking a multithreaded functional language with session types, Theor. Comput. Sci. 368 (2006) 64–87.

[53] N. Yoshida, P.M. Deniélou, A. Bejleri, R. Hu, Parameterised multiparty session types, in: C.H.L. Ong (Ed.), FOSSACS, volume 6014 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 128–145.

[54] N. Yoshida, V.T. Vasconcelos, Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication, Electr. Notes Theor. Comput. Sci. 171 (2007) 73–93.

[55] M. Yuang, Survey of protocol verification techniques based on finite state machine models, in: Computer Networking Symposium, 1988., Proceedings of the, pp. 164 –172. doi:`10.1109/CNS.1988.4993`.

Appendix A. Proofs

Lemma 1 (Coding)**.** For each type L in our language of local types, there exists a unique code $c(L) \in \mathbb{N}$.

Proof. We describe a way to assign natural numbers to local types. The coding is defined inductively; it assigns codes to the smallest components of local types, and then derives codes for larger types from the codes of their components. We use π_i to denote the i^{th} prime number and $c(t)$ for the code of component t.

We begin by assigning the codes $\pi_1, \pi_2, \ldots \pi_{26}$ to the characters of the English alphabet. Operators $!$, $?$, $;$, $\oplus$, $\otimes$, $\|$ are assigned the codes π_{27} to π_{32}, while π_{33} to π_{38} are used to code the n-ary versions of the operators, along with Kleene star and exponentiation, explained later. π_{39} is reserved for coding type subscript parameters, and π_{40} is used to code parenthesized types – both will be explained shortly. Beginning with π_{41}, we code the natural numbers: $c(0) = \pi_{41}$, $c(1) = \pi_{42}$, etc.

Actor names and message sorts are described by sequences of characters of the English alphabet and/or Arabic numerals. As such, we need to assign codes to their descriptions, henceforth simply referred to as *words*. The code of a word $w = w_1 w_2 \ldots w_n$ is given by

$$c(w) = \pi_1^{c(w_1)} \pi_2^{c(w_2)} \ldots \pi_n^{c(w_n)}$$

where w_i is the i^{th} character in the word. In the case of indexed words, (e.g., $client_i$) we have $w = (w_1 \ldots w_n)_{i_1 \ldots i_k}$ and

$$c(w) = \pi_1^{c(w_1)} \ldots \pi_n^{c(w_n)} \pi_{n+1}^{c(subscript)} \pi_{n+2}^{c(i_1)} \ldots \pi_{n+k+1}^{c(i_k)}$$

where $c(subscript) = \pi_{39}$ which was reserved above. Note that the number of indices in a simple type is known at compile time, as System-A does not allow for a parameterized number of parameters. Thus we have established unique codes for words.

Local types are formed from words combined with operators. We inductively assign codes to local types as follows:

$$
\begin{aligned}
c(\alpha!\mu) &= \pi_1^{c(\alpha)}\pi_2^{c(!)}\pi_3^{c(\mu)} \\
c(\alpha?\mu) &= \pi_1^{c(\alpha)}\pi_2^{c(?)}\pi_3^{c(\mu)} \\
c(L_1\ ;\ L_2\ ;\ \ldots\ ;\ L_k) &= \pi_1^{c(L_1)}\pi_2^{c(\ ;\ )}\pi_3^{c(L_2)}\pi_4^{c(\ ;\ )}\ldots\pi_{2k-2}^{c(\ ;\ )}\pi_{2k-1}^{c(L_k)} \\
c(L_1\ \oplus\ L_2\ \oplus\ \ldots\ \oplus\ L_k) &= \pi_1^{c(L_1)}\pi_2^{c(\ \oplus\ )}\pi_3^{c(L_2)}\pi_4^{c(\ \oplus\ )}\ldots\pi_{2k-2}^{c(\ \oplus\ )}\pi_{2k-1}^{c(L_k)} \\
c(L_1\ \otimes\ L_2\ \otimes\ \ldots\ \otimes\ L_k) &= \pi_1^{c(L_1)}\pi_2^{c(\ \otimes\ )}\pi_3^{c(L_2)}\pi_4^{c(\ \otimes\ )}\ldots\pi_{2k-2}^{c(\ \otimes\ )}\pi_{2k-1}^{c(L_k)} \\
c(L_1\ \|\ L_2\ \|\ \ldots\ \|\ L_k) &= \pi_1^{c(L_1)}\pi_2^{c(\ \|\ )}\pi_3^{c(L_2)}\pi_4^{c(\ \|\ )}\ldots\pi_{2k-2}^{c(\ \|\ )}\pi_{2k-1}^{c(L_k)}
\end{aligned}
$$

where α, μ are—in this context—meta symbols denoting an actor name and a message sort respectively. Note that the number of arguments of the operators is fixed above. The codes for the parameterized versions are:

$$
\begin{aligned}
c\Big(\mathop{\Large\circledcirc}_{\iota=\chi}^{\nu}\Big) &= \pi_1^{\pi_{33}}\pi_2^{c(\iota)}\pi_3^{c(\chi)}\pi_4^{c(\nu)} \\
c\Big(\mathop{\Large\bigoplus}_{\iota=\chi}^{\nu}\Big) &= \pi_1^{\pi_{34}}\pi_2^{c(\iota)}\pi_3^{c(\chi)}\pi_4^{c(\nu)} \\
c\Big(\mathop{\Large\bigotimes}_{\iota=\chi}^{\nu}\Big) &= \pi_1^{\pi_{35}}\pi_2^{c(\iota)}\pi_3^{c(\chi)}\pi_4^{c(\nu)} \\
c\Big(\mathop{\Large\|}_{\iota=\chi}^{\nu}\Big) &= \pi_1^{\pi_{36}}\pi_2^{c(\iota)}\pi_3^{c(\chi)}\pi_4^{c(\nu)}
\end{aligned}
$$

where ι, χ, ν are meta-symbols denoting an index name, a starting value (integer) and a type parameter respectively. As a reminder, π_{33} to π_{36} had been reserved for this purpose. Coding local types that use parameterized operators is done as follows:

$$
c\Big(\mathop{OP}_{\iota=\chi}^{\nu} L_\iota\Big) = \pi_1^{\,c\left(\mathop{OP}_{\iota=\chi}^{\nu}\right)}\pi_2^{c(L_\iota)}
$$

where OP denotes any of $\circledcirc$, $\oplus$, $\otimes$, $\|$. Since L_ι is parameterized by ι, the rules for lexicographic coding of words with subscripts take care of this.

Similarly,

$$c(G^*) = \pi_1^{\pi_{37}} \pi_2^{c(G)}$$
$$c(G^\nu) = \pi_1^{\pi_{38}} \pi_2^{c(\nu)} \pi_3^{c(G)}$$

where ν is a meta-symbol ranging over words that describe type parameters. Recall that π_{37} and π_{38} had been reserved for the purpose above. Parenthesized types are coded as

$$c\big((L)\big) = \pi_1^{\pi_{40}} \pi_2^{c(L)}$$

where π_{40} was reserved for this purpose.

It is clear from the way we use prime numbers that the fundamental theorem of arithmetic (uniqueness of prime factorization) implies the uniqueness of codes assigned to local types. □

Appendix B. Local Type Reduction Semantics

Table B.12: *The semantics of Local Types, in close accordance to the semantics of Lang-A in Table 4. The rules transform configurations* $(SEL, SHUF, T, M, \Delta, \Pi)$*, where SEL and SHUF are partial functions from markers to values; T is the trace produced so far; M is the multiset of pending messages;* Δ *is the environment seen so far; and* Π *is the multiset of executing actors* $\langle L \rangle_a$ *where a is the actor executing the behavior specified by* $L \in \mathscr{L}$*. Unchanged elements are omitted from the rules. Function* $v(\cdot)$ *evaluates expressions without side-effects;* $[\cdot/\cdot]$ *denotes substitution of free variables;* $[\cdot \mapsto \cdot]$ *updates functions point-wise; for an explanation of the various markers used here, see Section 5.2.*

$$\text{(LS-Init)} \quad \Delta \cup \{L : a\},\, \Pi \;\rightsquigarrow\; \Delta,\, \Pi \cup \langle L\,;\,\tau \rangle_a$$

$$\text{(LS-Seq)}\frac{L_1 \;\rightsquigarrow\; L_1'}{L_1\,;\,L_2 \;\rightsquigarrow\; L_1'\,;\,L_2}$$

$$\text{(LS-Empty)}\ \frac{}{\tau\,;\,L \;\rightsquigarrow\; L} \qquad \text{(LS-Exp)}\ \frac{}{L^n \;\rightsquigarrow\; \overset{n}{\underset{i=1}{\circledcirc}} L} \qquad \text{(LS-Star)}\ \frac{}{L^* \;\rightsquigarrow\; (L\,;\,L^*) \,\oplus\, \tau}$$

$$\text{(LS-SeqN)}\ \frac{v(n_1) \le v(n_2)}{\overset{n_2}{\underset{i=n_1}{\circledcirc}} L \;\rightsquigarrow\; L[v(n_1)]/i]\,;\,\big(\overset{n_2}{\underset{i=v(n_1+1)}{\circledcirc}} L\big)} \qquad \text{(LS-SeqNil)}\ \frac{v(n_1) > v(n_2)}{\overset{n_2}{\underset{i=n_1}{\circledcirc}} L \;\rightsquigarrow\; \tau}$$

$$\text{(LS-AuxBegin)}\ \frac{x = v(n_2) - v(n_1) + 1, \quad OP \in \{\bigoplus, \bigotimes, \|\}, \quad k \text{ fresh}}{\Pi \cup \big\langle \overset{n_2}{\underset{i=n_1}{OP}} L\,;\,L' \big\rangle_a^{sel,\,shuf} \;\rightsquigarrow\; \Pi \cup \big\langle \overset{n_2}{\underset{i=n_1}{OP}} L\,;\,L' \big\rangle_a^{sel,\,shuf,\,sp_k^x}}$$

$$\text{(LS-AuxEnd)}\frac{v(n_1) > v(n_2), \quad x > 0, \quad OP \in \{\bigoplus, \bigotimes, \|\}}{\Pi \cup \big\{ \big\langle \overset{n_2}{\underset{i=n_1}{OP}} L\,;\,L' \big\rangle_a^{sel,\,shuf,\,sp_k^x} \big\} \;\rightsquigarrow\; \Pi}$$

Continued on the next page.

Table B.12: *The semantics of local types. Continued from the previous page.*

(LS-AuxNil)
$$\frac{x \leq 0, \quad OP \in \{\bigoplus, \bigotimes, \|\}}{\Big\langle \mathop{OP}_{i=n_1}^{n_2} L \,;\, L' \Big\rangle_a^{sel,\, shuf,\, sp_k^x} \rightsquigarrow \Pi \cup \big\langle L' \big\rangle_a^{sel,\, shuf}}$$

(LS-Paral)
$$\Pi \cup \big\langle (L_1 \parallel L_2) \,;\, L_3 \big\rangle_a^{sel,\, shuf} \rightsquigarrow \Pi \cup \Big\{ \big\langle L_1 \,;\, j_k^2 \,;\, L_3 \big\rangle_a^{sel,\, shuf}, \big\langle L_2 \,;\, j_k^2 \,;\, L_3 \big\rangle_a^{sel,\, shuf} \Big\} \qquad k \text{ fresh}$$

(LS-ParalN)
$$\frac{v(n_1) \leq v(n_2)}{\Big(\Pi \cup \Big\{ \Big\langle \mathop{\|}_{i=n_1}^{n_2} L \,;\, L' \Big\rangle_a^{sel,\, shuf,\, sp_k^x} \Big\}\Big) \rightsquigarrow \Big(\Pi \cup \Big\{ \big\langle L[v(n_1)/i] \,;\, j_k^x \,;\, L' \big\rangle_a^{sel,\, shuf}, \Big\langle \mathop{\|}_{i=v(n_1+1)}^{n_2} L \,;\, L' \Big\rangle_a^{sel,\, shuf,\, sp_k^x} \Big\}\Big)}$$

(LS-Join)
$$\Pi \cup \Big\{ \underbrace{\big\langle j_k^x \,;\, L \big\rangle_a^{sel,\, shuf}, \ldots, \big\langle j_k^x \,;\, L \big\rangle_a^{sel,\, shuf}}_{x \text{ instances}} \Big\} \rightsquigarrow \Pi \cup \big\langle L \big\rangle_a^{sel,\, shuf}$$

(LS-Send)
$$\big(M,\, \Pi \cup \big\langle b!m \,;\, L \big\rangle_a^{sel,\, shuf}\big) \rightsquigarrow \big(M \cup (m)_{a,b},\, \Pi \cup \big\langle L \big\rangle_a^{sel,\, shuf}\big)$$

(LS-Recv)
$$\frac{\begin{array}{c} SEL(x) \in \{\bot,\, sel(x)\}\, \forall x \in dom(sel) \\ SHUF(y) \in \{\bot,\, shuf(y)\}\, \forall y \in dom(shuf) \end{array}}{\begin{array}{r} \big(SEL,\, SHUF,\, T,\, M \cup (m)_{a,b},\, \Pi \cup \big\langle a?m \,;\, L \big\rangle_b^{sel,\, shuf}\big) \rightsquigarrow \\ \big(SEL[x \mapsto sel(x) \,|\, x \in dom(sel)], \\ SHUF[x \mapsto shuf(x) \,|\, x \in dom(shuf)], \\ T \cdot (a \xrightarrow{t} b),\, M,\, \Pi \cup \big\langle L \big\rangle_b^{sel,\, shuf}\big) \end{array}}$$

Continued on the next page.

Table B.12: *The semantics of local types. Continued from the previous page.*

(LS-Choice)
$$\Pi \cup \big\langle L_1 \oplus L_2) \,;\, L_3 \big\rangle_a^{sel,\, shuf} \rightsquigarrow \Pi \cup \Big\{ \big\langle L_1 \,;\, L_3 \big\rangle_a^{sel[k \mapsto 1],\, shuf} \big\langle L_2 \,;\, L_3 \big\rangle_a^{sel[k \mapsto 2],\, shuf} \Big\} \qquad k \text{ fresh}$$

(LS-ChoiceDone)
$$\frac{\exists x \in dom(sel) : SEL(x) \notin \{\bot,\, sel(x)\}}{\big(SEL,\, \Pi \cup \big\langle L \big\rangle_a^{sel,\, shuf}\big) \rightsquigarrow \big(SEL,\, \Pi\big)}$$

(LS-ChoiceN)
$$\frac{v(n_1) \leq v(n_2)}{\Pi \cup \Big\{ \big\langle \bigoplus_{i=n_1}^{n_2} L \,;\, L' \big\rangle_a^{sel,\, shuf,\, sp_k^x} \Big\} \rightsquigarrow \Pi \cup \Big\{ \big\langle L[v(n_1)/i] \,;\, L' \big\rangle_a^{sel[k \mapsto v(n_1)],\, shuf}, \big\langle \bigoplus_{i=v(n_1+1)}^{n_2} L \,;\, L' \big\rangle_a^{sel,\, shuf,\, sp_k^x} \Big\}}$$

(LS-Shuffle)
$$\Pi \cup \big\langle (L_1 \otimes L_2) \,;\, L_3 \big\rangle_a^{sel,\, shuf} \rightsquigarrow \Pi \cup \Big\{ \big\langle L_1 \,;\, u_k \,;\, j_k^2 \,;\, L_3 \big\rangle_a^{sel,\, shuf[k \mapsto 1]}, \big\langle L_2 \,;\, u_k \,;\, j_k^2 \,;\, L_3 \big\rangle_a^{sel,\, shuf[k \mapsto 2]} \Big\} \qquad k \text{ fresh}$$

(LS-ShuffleDone)
$$\big(SHUF,\, \Pi \cup \big\{ \big\langle u_k \,;\, j_k^x \,;\, L \big\rangle_a^{sel,\, shuf} \big\}\big) \rightsquigarrow SHUF[k \mapsto \bot],\, \Pi \cup \big\langle j_k^x \,;\, L \big\rangle_a^{sel,\, shuf[k \mapsto \bot]}$$

(LS-ShuffleN)
$$\frac{v(n_1) \leq v(n_2)}{\Pi \cup \Big\{ \big\langle \bigotimes_{i=n_1}^{n_2} L \,;\, L' \big\rangle_a^{sel,\, shuf,\, sp_k^x} \Big\} \rightsquigarrow \Pi \cup \Big\{ \big\langle L[v(n_1)/i] \,;\, u_k \,;\, j_k^x \,;\, L' \big\rangle_a^{sel,\, shuf[k \mapsto v(n_1)]}, \big\langle \bigotimes_{i=v(n_1+1)}^{n_2} L \,;\, L' \big\rangle_a^{sel,\, shuf,\, sp_k^x} \Big\}}$$