Representing Session Types

Peter Brottveit Bock, Alessandro Bruni,
Agata Murawska, and Carsten Schürmann

IT University of Copenhagen, Denmark
{pbrb, brun, agmu, carsten}@itu.dk

Abstract

In this paper we propose a logical foundation of processes and their focused normal forms. We use a linear meta-language based on substructural operational semantics to describe focused forms of processes, and compare them to standard π-calculus processes with their respective operational semantics.

The overall goal of this research is to understand how to reason about processes, multi-party communication and global types, and how to mechanize properties such as deadlock freeness and liveness. We are also interested in establishing the limitations of this approach.

1 Introduction

The π-calculus was developed by Robin Milner [Mil99] as a language to describe processes and concurrent systems. Session types [HVK98] soon followed as a way to rule out nonsensical processes, e.g. ones where send and receive actions don't match. Even though the intuition was there from quite early on [**?**], it took over a decade until the seminal work by Caires and Pfenning [CP10] provided a logical interpretation of π-calculus. Indeed, there is a Curry-Howard correspondence between processes and linear logic, just as there is such a relation between λ-calculus and intuitionistic logic.

Recently much work has been conducted in this area [CPPT13, Wad14, CMS14, LM16], predominantly using logic to explain the π-calculus, deadlock freedom and session fidelity. A lot less attention has been given to approaching the topic from the opposite side, striving to derive new process calculi and their respective semantics from the logics. In this paper, we do exactly this: we use a focused formulation of linear logic, combine it with forward chaining cut-elimination, and arrive at a focused process algebra.

A technical contribution of our work is a case study of using a logical framework to encode (focused) session types. We use a variant of the concurrent logical framework CLF [WCPW02] to describe process calculi using substructural operational semantics (SSOS) [PS09]. Our formulations are surprisingly elegant, and the notion of trace equivalence provided by the logical framework represents commutative conversions adequately. A corollary of this observation is that, when studying process algebras that are agnostic to the different interleavings of several processes executing in parallel, we no longer have to worry about modeling interleavings – on the trace level, these are captured by the equivalences provided by the logical framework.

The remainder of this paper is structured as follows. In Section 2, we give a brief introduction to process algebra, the π-calculus, and different forms of operational semantics. In Section 3 we direct our attention to SSOS and describe a variant of CLF that we will use in the subsequent sections to derive process algebras and their respective operational semantics from logic. We will look at two process algebras, one derived from standard, unfocused linear logic in Section 4, the other reconstructed from focused linear logic in Section 5. We assess results and conclude in Section 6.

2 Process Algebra

We begin this section by introducing our variant of the π-calculus and its operational semantics. We then move to discuss a typed version of the process algebra and show how the well-typed processes belonging to principal cuts reduce to well-typed processes, which are the result of such cuts.

2.1 Two-Layer Typed Processes

We present the two-layered system of session typed processes. The bottom layer is made out of standard π-calculus processes, whereas the upper layer is a sequence of definitions with a final process.

Channels	α	::=	$\odot \mid x$
Session Types	A, B	::=	$a \mid 1 \mid A \otimes B \mid A \multimap B \mid A \,\&\, B \mid A \oplus B$
Contexts	Δ, Γ	::=	$\cdot \mid \Delta, x : A$
Processes	P, Q	::=	$\mathbf{fwd}_\alpha\ x \mid \mathbf{end}_\alpha \mid \mathbf{wait}^x.P \mid \nu x^A.P \mid (P \mid Q)$
			$\mid \alpha\langle y\rangle.P \mid \alpha(y).P \mid \alpha.\mathbf{case}\ (P, Q) \mid \alpha.\mathbf{inl}; P \mid \alpha.\mathbf{inr}; P \mid \mathbf{nil}$
Extended Processes	E, F	::=	$P \mid \mathbf{let}\ x := P\ \mathbf{in}\ E$
Transition Labels	l	::=	$\tau \mid \overline{x\langle y\rangle} \mid \underline{x(y)} \mid \overline{\nu y.x\langle y\rangle} \mid x.\mathbf{inl} \mid x.\mathbf{inr} \mid \overline{x.\mathbf{inl}} \mid \overline{x.\mathbf{inr}}$
			$\mid x.\mathbf{end} \mid \overline{x.\mathbf{end}}$

Where a simple typed process ensures a specific behavior while making some assumptions about the environment in which it is executed, an extended process *extends* this concept into a family of processes. When dealing with a single process P we abstract away the channel on which it operates, which we denote as $\odot$. We reintroduce this channel when we connect P to the environment E: in $\mathbf{let}\ x := P\ \mathbf{in}\ E$ process P operates on channel x.

The session types correspond to our fragment of linear logic, and include atoms and the unit type, the multiplicative and the additive fragments of linear logic. A context provides an association between channel names their respective session types.

The process $\mathbf{fwd}_\alpha\ x$ forwards messages between the channels α and x, process $\mathbf{end}_\alpha$ signals the end of communication on α, process $\mathbf{wait}^x P$ waits for the end of communication on channel x and proceeds as P, $\nu x^A.P$ restricts the scope of x of type A within P, $P \mid Q$ is the parallel composition of P and Q, $\alpha\langle y\rangle.P$ sends y on α and proceeds with P, $\alpha(y).P$ receives y on α and proceeds with P, $\alpha.\mathbf{case}\ (P, Q)$ offers to continue either as P or Q depending on the choice made on α, whereas $\alpha.\mathbf{inl}; P$ and $\alpha.\mathbf{inr}; P$ signals the choice of either the left or right behavior of α. Finally, the process $\mathbf{nil}$ represents a terminated process.

In the extend process E, a single P represents the final process to be executed. We expect this process to be of type 1. Simple processes are incorporated into an extended process E via a let-compositions $\mathbf{let}\ x := P\ \mathbf{in}\ E$, where process P is now connected to x for E to consume.

Finally we have transition labels l: τ denotes an internal action; $\overline{x\langle y\rangle}$ denotes an output of y over x; $\underline{x(y)}$ denotes an input of y over x; $\overline{\nu y.x\langle y\rangle}$ denotes scope extrusion, where the scope of a restricted channel y is extended by sending it over x; $x.\mathbf{inl}$ and $x.\mathbf{inr}$ are respectively the left choice and the right choice of case over channel x; and conversely $\overline{x.\mathbf{inl}}$ and $\overline{x.\mathbf{inr}}$ request respectively the left and the right projection of a case; finally, $x.\mathbf{end}$ and $\overline{x.\mathbf{end}}$ respectively denote a process waiting for the end of communication on x and a process ending all communication on x.

2.2 Semantics

Semantic rules:

$$\boxed{P \xrightarrow{l} Q} \quad \text{Process } P \text{ steps to process } Q \text{ with a trace } l$$

$$\frac{P \xrightarrow{l} Q}{\nu x.P \xrightarrow{l} \nu x.Q}\ \text{res} \qquad \frac{P \xrightarrow{l} Q}{P \mid R \xrightarrow{l} Q \mid R}\ \text{par} \qquad \frac{P \xrightarrow{\bar{l}} P' \quad Q \xrightarrow{l} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{com}$$

$$\frac{P \xrightarrow{\overline{\nu y.x\langle y\rangle}} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} \nu y.(P' \mid Q')}\ \text{close} \qquad \frac{P \xrightarrow{\overline{x\langle y\rangle}} Q}{\nu y.P \xrightarrow{\overline{\nu y.x\langle y\rangle}} Q}\ \text{open} \qquad \frac{}{x\langle y\rangle P \xrightarrow{\overline{x\langle y\rangle}} P}\ \text{out}$$

$$\frac{}{x(y)P \xrightarrow{x(z)} P\{^z/_y\}}\ \text{in} \qquad \frac{}{x.\mathbf{case}\ (P,Q) \xrightarrow{x.\mathbf{inl}} P}\ \text{csl} \qquad \frac{}{x.\mathbf{case}\ (P,Q) \xrightarrow{x.\mathbf{inr}} Q}\ \text{csr}$$

$$\frac{}{x.\mathbf{inl}; P \xrightarrow{\overline{x.\mathbf{inl}}} P}\ \text{inl} \qquad \frac{}{x.\mathbf{inr}; P \xrightarrow{\overline{x.\mathbf{inr}}} P}\ \text{inr} \qquad \frac{}{\mathbf{end}_x \xrightarrow{\overline{x.\mathbf{end}}} \mathbf{nil}}\ \text{end} \qquad \frac{}{\mathbf{wait}^x.P \xrightarrow{x.\mathbf{end}} P}\ \text{wait}$$

$$\boxed{E \xrightarrow{l} F} \quad \text{Extended process } E \text{ steps to extended process } F \text{ with a trace } l$$

$$\frac{\mathbf{fwd}_\odot\ x \mid P \xrightarrow{l} \mathbf{fwd}_\odot\ x \mid P'}{\mathbf{let}\ x := P\ \mathbf{in}\ E \xrightarrow{l} \mathbf{let}\ x := P'\ \mathbf{in}\ E} \qquad \frac{E \xrightarrow{l} E'}{\mathbf{let}\ x := P\ \mathbf{in}\ E \xrightarrow{l} \mathbf{let}\ x := P\ \mathbf{in}\ E'}$$

$$\frac{\mathbf{fwd}_\odot\ x \mid P \xrightarrow{\bar{l}} \mathbf{fwd}_\odot\ x \mid P' \quad E \xrightarrow{l} E'}{\mathbf{let}\ x := P\ \mathbf{in}\ E \xrightarrow{\tau} \mathbf{let}\ x := P'\ \mathbf{in}\ E'}$$

Congruence rules:

$$P \mid \mathbf{nil} \equiv P \quad \text{(SNil)} \qquad P \mid Q \equiv Q \mid P \quad \text{(SParC)} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad \text{(SParA)}$$
$$\nu x.\mathbf{nil} \equiv \mathbf{nil} \quad \text{(SResNil)} \qquad \nu x.\nu y.P \equiv \nu y.\nu x.P \quad \text{(SResC)} \qquad P \equiv_\alpha Q \Rightarrow P \equiv Q \quad \text{(SRen)}$$
$$x \notin fn(P) \Rightarrow P \mid \nu x.Q \equiv \nu x.(P \mid Q) \quad \text{(SScope)} \qquad \mathbf{fwd}_y\ x \mid P \equiv P\{^y/_x\} \quad \text{(SFwd)}$$
$$\mathsf{let}\ x := P\ \mathsf{in}\ \mathsf{let}\ y := Q\ \mathsf{in}\ E \equiv \mathsf{let}\ y := Q\ \mathsf{in}\ \mathsf{let}\ x := P\ \mathsf{in}\ E \quad \text{(SLetFloat)}$$

Figure 1: Semantic and congruence rules of our calculus

Figure 1 presents the rules defining the labeled semantics and congruence relations of our π-calculus, where we omit type annotations. The relation $P \equiv Q$ is the reflexive, symmetric, transitive and congruent closure generated by the given rules. The following side conditions apply: rule (res) requires $x \notin fn(l)$; rule (par) requires $bn(l) \cap fn(R) = \emptyset$; rule (close) requires $y \notin fn(Q)$. A semantic step $P \xrightarrow{l} Q$ (resp. $E \xrightarrow{l} F$) is the smallest relation generated by the given rules and congruence rules for P and Q (resp. E and F).

Note that we do not communicate under prefixes, which is in line with standard semantics of π-calculus. From the perspective of cut-elimination in logic, this is non-standard.

2.3 Type system

Figure 2 shows the session type system for the processes defined in Section 2.1. The type judgment is on the form $P \triangleright \Delta \vdash A$, where P is the process providing behavior A on the channel $\odot$, consuming the channels declared in Δ. The typing rules correspond to linear intuitionistic logic, and they are the same as Caires and Pfenning's [CP10], but differ in that the channel in

$\boxed{P \rhd \Delta \vdash C}$ Process P provides a behavior specified by type C using variables with their behavior declared in Δ

$$\frac{}{\mathbf{fwd}_{\odot}\ x \rhd x : C \vdash C}\ ax \qquad \frac{P \rhd \Gamma \vdash A \qquad Q \rhd \Delta, x : A \vdash C}{\nu x^{A}.(P \mid Q) \rhd \Gamma, \Delta \vdash C}\ cut$$

$$\frac{}{\mathbf{end}_{\odot} \rhd \vdash 1}\ 1_R \qquad \frac{P \rhd \Delta \vdash C}{\mathbf{wait}^{x}.P \rhd \Delta, x : 1 \vdash C}\ 1_L$$

$$\frac{P \rhd \Gamma \vdash A \quad Q \rhd \Delta \vdash B}{\nu y^{A}.\odot\langle y\rangle.(P \mid Q) \rhd \Gamma, \Delta \vdash A \otimes B}\ \otimes_R \qquad \frac{P \rhd \Delta, y : A, x : B \vdash C}{x(y).P \rhd \Delta, x : A \otimes B \vdash C}\ \otimes_L$$

$$\frac{P \rhd \Delta, y : A \vdash B}{\odot(y).P \rhd \Delta \vdash A \multimap B}\ \multimap_R \qquad \frac{P \rhd \Delta \vdash A \quad Q \rhd \Gamma, x : B \vdash C}{\nu y^{A}.x\langle y\rangle.(P \mid Q) \rhd \Delta, \Gamma, x : A \multimap B \vdash C}\ \multimap_L$$

$$\frac{P \rhd \Delta \vdash A \quad Q \rhd \Delta \vdash B}{\odot.\mathbf{case}\ (P, Q) \rhd \Delta \vdash A\ \&B}\ \&_R \qquad \frac{P \rhd \Delta, x : A \vdash C \quad Q \rhd \Delta, x : B \vdash C}{x.\mathbf{case}\ (P, Q) \rhd \Delta, x : A \oplus B \vdash C}\ \oplus_L$$

$$\frac{P \rhd \Delta, x : A \vdash C}{x.\mathbf{inl}; P \rhd \Delta, x : A\ \&B \vdash C}\ \&_{L_1} \qquad \frac{Q \rhd \Delta, x : B \vdash C}{x.\mathbf{inr}; Q \rhd \Delta, x : A\ \&B \vdash C}\ \&_{L_2}$$

$$\frac{P \rhd \Delta \vdash A}{\odot.\mathbf{inl}; P \rhd \Delta \vdash A \oplus B}\ \oplus_{R_1} \qquad \frac{Q \rhd \Delta \vdash B}{\odot.\mathbf{inr}; Q \rhd \Delta \vdash A \oplus B}\ \oplus_{R_2}$$

$\boxed{\Delta \vdash E : \Gamma}$ Extended process E terminates using behavior abstractions declared in context Γ

$$\frac{P \rhd \Gamma \vdash 1}{\Gamma \vdash P} \qquad \frac{P \rhd \Gamma_1 \vdash A \quad \Gamma_2, x : A \vdash E}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\ x := P\ \mathbf{in}\ E}$$

Figure 2: Typing for Two-Layer Processes

the conclusion is not named.

2.4 Cut reductions in π-calculus

In this section we show how possible cuts in the type system presented in Section 2.3 can be described as concurrent processes, and how a semantic reduction step corresponds to the application of the correspondent cut reduction, transforming well-typed processes into well-typed reduced processes.

The application of a cut reduction requires that a formula on the right hand side of the sequent be matched with a formula on the left hand side, which corresponds to a channel. Hence the processes here take the form $\nu x.(\mathbf{fwd}_{\odot}\ x \mid P \mid Q)$ where P is a process operating on the standard channel $\odot$, Q expects communication to happen on x, and the two are linked by the forwarder $\mathbf{fwd}_{\odot}\ x$ which corresponds to the application of the axiom rule in the logic.

$\nu x^{A}.(\mathbf{fwd}_{\odot}\ x \mid \mathbf{end}_{\odot} \mid \mathbf{wait}^{x}.P) \xrightarrow{\tau} \nu x^{A}.(\mathbf{fwd}_{\odot}\ x \mid \mathbf{nil} \mid P)$
$\nu x^{A\otimes B}.(\mathbf{fwd}_{\odot}\ x \mid \nu y^{A}.\odot\langle y\rangle.(P \mid Q) \mid x(y).R) \xrightarrow{\tau} \nu x^{B}.\nu y^{A}.(\mathbf{fwd}_{\odot}\ x \mid P \mid Q \mid R)$
$\nu x^{A\multimap B}.(\mathbf{fwd}_{\odot}\ x \mid \odot(y).P \mid \nu y^{A}.x\langle y\rangle.(Q \mid R)) \xrightarrow{\tau} \nu x^{B}.\nu y^{A}.(\mathbf{fwd}_{\odot}\ x \mid P \mid Q \mid R)$
$\nu x^{A\ \&B}.(\mathbf{fwd}_{\odot}\ x \mid \odot.\mathbf{case}\ (P,Q) \mid x.\mathbf{inl}; R) \xrightarrow{\tau} \nu x^{A}.(\mathbf{fwd}_{\odot}\ x \mid P \mid R)$
$\nu x^{A\ \&B}.(\mathbf{fwd}_{\odot}\ x \mid \odot.\mathbf{case}\ (P,Q) \mid x.\mathbf{inr}; R) \xrightarrow{\tau} \nu x^{B}.(\mathbf{fwd}_{\odot}\ x \mid Q \mid R)$
$\nu x^{A\oplus B}.(\mathbf{fwd}_{\odot}\ x \mid \odot.\mathbf{inl}; P \mid x.\mathbf{case}\ (Q,R)) \xrightarrow{\tau} \nu x^{A}.(\mathbf{fwd}_{\odot}\ x \mid P \mid Q)$
$\nu x^{A\oplus B}.(\mathbf{fwd}_{\odot}\ x \mid \odot.\mathbf{inr}; P \mid x.\mathbf{case}\ (Q,R)) \xrightarrow{\tau} \nu x^{B}.(\mathbf{fwd}_{\odot}\ x \mid P \mid R)$

3 The CLF Framework

In this section we give a brief overview about a framework based on substructural operational semantics that we will use as a meta-language to describe concurrent systems. In logical frameworks based on the dependently typed λ-calculus, the prevailing method for representing judgments are types, and derivations are correspondingly represented as objects. A rule with premises $\mathcal{J}_1 \dots \mathcal{J}_n$ and conclusion $\mathcal{J}$ is represented by constants in the type theory, mapping inhabitants of $\mathcal{J}_1 \dots \mathcal{J}_n$ into an inhabitant of $\mathcal{J}$. Although extremely elegant, this way of representing works best in the context of derivation trees, prevalent in logic, type systems, and operational semantics, where we can establish the adequacy of the encoding, meaning that we can show that every derivation tree has a unique representation in the logical framework and that each canonical form in the logical framework corresponds to a valid derivation. Formally, we call a representation *adequate*, if there is a bijection between *individual* derivations and objects.

The situation is quite different for representations of concurrent systems. Here, our adequacy result should not identify individual derivation trees and objects in the framework, but instead identify objects in the type theory with an entire equivalence class of derivations. This is a subtle but important difference. Consider for example the representation of an operational semantics of a programming language with non-deterministic choice. In such a language, the result of a computation might be not be deterministically determined, in fact, each possible trace of the operational semantics is represented as its own object, and these objects are in general not equivalent (modulo α, β, and η conversions). In process algebra, however, we deal with message exchanges. A trace records all the messages sent and received by the different parties, and it is natural to consider two traces equivalent, if messages that do not depend on each other, are exchanged. Equivalence classes, that have multiple manifestations, for example, reorderings, are also called concurrent objects, and their representation is called *adequate*, if there there is a bijection between the *equivalence classes* and an appropriately typed object of the type theory.

The CLF framework is implemented as the logical framework Celf, which is available for download from `www.github.com/carstenschuermann/celf`. We use it to describe the representations of concurrent systems throughout this paper.

3.1 Syntax

We begin now with the presentation of the meta-language to describe the meta-language of concurrent systems. Our presentation follows closely [SN11], which essentially describes a focused system for linear logic. On the formula level, we distinguish negative and positive formulas, where p denotes a pattern, and N a *normal* object that we define below. The logical framework

that we consider here has only

$$\begin{array}{rrcl} \text{Atomic Formulas} & P & ::= & a \mid P\ N \\ \text{Negative Formulas} & A^- & ::= & \Pi p : A^+.B^- \mid A^- \& B^- \mid \{A\} \mid P \\ \text{Positive Formulas} & A^+ & ::= & \exists p : A^+.B^+ \mid \downarrow A^- \mid\ !A^- \mid 1. \end{array}$$

The notation $\{\cdot\}$ used for defining negative formulas describes a polarity shift from positive to negative formulas. For convenience, we have omitted notation for the inverse polarity shift, which can always be derived. A pattern accounts for an inversion phase on the left following the structure of a positive formula and is defined as follows

$$\text{Patterns} \quad p \quad ::= \quad \langle\langle p_1, p_2 \rangle\rangle \mid \downarrow x \mid\ !x \mid 1.$$

Next, we focus the presentation on the object level.

We write N for normal forms of negative formulas, M for normal forms or positive formulas, and E for expressions. To simplify notation, we introduce atomic objects O. All three syntactic categories are defined as follows:

$$\begin{array}{rrcl} \text{Atomic Objects} & O & ::= & x \mid c \mid O\ M \mid \pi_1 O \mid \pi_2 O \\ \text{Normal Objects} & N & ::= & \lambda p.N \mid \langle N_1, N_2 \rangle \mid \text{let } E \text{ in } \{M\} \mid O \\ \text{Trace Objects} & E & ::= & \cdot \mid p = O; E \\ \text{Monadic Objects} & M & ::= & \langle\langle M_1, M_2 \rangle\rangle \mid \downarrow N \mid\ !N \mid 1. \end{array}$$

Before we declare the judgments, we need to introduce typing contexts for linear and unrestricted variables

$$\begin{array}{rrcl} \text{Linear Contexts:} & \Delta & ::= & \cdot \mid \Delta, u : A \\ \text{Unrestricted Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

and signatures, for which we write Σ, which declare type families and constant symbols with their respective types. If convenient, we use a mixed contexts instead where we write $:$ for a declaration in the unrestricted and $\hat{:}$ for a declaration in the linear context.

$$\text{Mixed Contexts:} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A \mid \Gamma, u \hat{:} A$$

In the interest of space, we do not discuss the kind level in detail, but refer instead the interested reader to the relevant literature [SN11]. In a slight derivation from this presentation, we present the framework without spines, but instead with an explicit category for traces, following [CPS+12]. Note, that in Celf syntax, uppercase variables are implicitly Π-abstracted.

3.2 Static Semantics

In this section we describe the typing judgments that define the static semantics of our framework. We begin with the discussion of patterns, as this is the only judgment that is not mutually dependent with others. Patterns correspond to maximal focused phases on the left.

$$\Gamma; \Delta; p : A^+ \vdash \Gamma'; \Delta' \quad \text{(Pattern decomposition)}$$

The rules defining the judgment are depicted in Figure 3. The simplest cases of patterns are those of unit, a linear or unrestricted variable subject to instantiation, ending the focus phase. The rule $\exists P$ represents dependent pairing.

$$\frac{\Gamma;\Delta;p_1 : A^+ \vdash \Gamma';\Delta' \quad \Gamma';\Delta';p_2 : B^+ \vdash \Gamma'';\Delta''}{\Gamma;\Delta;\langle\langle p_1, p_2\rangle\rangle : \exists p_1 : A^+.B^+ \vdash \Gamma'';\Delta''}\ \exists P \qquad \frac{}{\Gamma;\Delta;1 : 1 \vdash \Gamma;\Delta}\ 1P$$

$$\frac{}{\Gamma;\Delta;\downarrow x : \downarrow A^- \vdash \Gamma;(\Delta, x : A^-)}\ \downarrow P \qquad \frac{}{\Gamma;\Delta;!x : !A^- \vdash (\Gamma, x : A^-);\Delta}\ !P$$

Figure 3: Pattern Decomposition

The remaining judgments define the static semantics of the various categories of objects, including atomic, normal, trace, and monadic objects.

$$\begin{array}{ll} \Gamma;\Delta \vdash O \Rightarrow A^- & \text{(Typing atomic objects)} \\ \Gamma;\Delta \vdash N \Leftarrow A^- & \text{(Typing normal objects)} \\ \Gamma;\Delta \vdash M \Leftarrow A^+ & \text{(Typing monadic objects)} \\ \Gamma;\Delta \vdash E \Leftarrow \Gamma';\Delta' & \text{(Typing trace objects)} \end{array}$$

The judgments are mutually recursive and defined in the 4. The rules are presented as as focused system and they are largely standard. We omit the precise definition of hereditary substitutions used in rules *app* and $\exists I$ and written as $[M/p]A$ from this presentation in the interest of space and because it is standard. Hereditary substitutions decomposes object M following the structure given by pattern p, replaces variables by terms and normalizes the result, all in one go. The details describing hereditary substitutions can be found in [SN11].

We only comment on trace objects. A trace

$$p_1 = O_1; \ldots p_i = O_i; \ldots p_n = O_n; \cdot$$

can be seen as a sequence of rewrite steps, rewriting a context pair

$$\Gamma_0;\Delta_0 \ldots \Gamma_i;\Delta_i \ldots \Gamma_n;\Delta_n.$$

Here, each O_i describes which rule to apply, the head of O_i either refers to a constant c, the name of a statically defined rule declared in the signature, or to a variable x, the name of a dynamically defined rule, introduced by a previous step. The arguments in O_i can be seen as the parameters to the rule, and the pattern p_i results assigns names to the new introduced assumptions in the context.

Example 3.1. Let us define the following signature, which is intended to count the number of tokens represented in a linear context. We use Celf to describe the signature.

```
% Natural Numbers
nat : type.
z : nat.
s : nat → nat.

% Counting Predicate
count : nat → type.
token : type.

% Counting Rules
rule : token ⊸ count N ⊸ {count (s N)}.
```

In addition, let our linear context contain three `token`s and the predicate `count z`. Starting from the initial context pair $\Gamma_0;\Delta_0$:

$$\cdot;\cdot, u_1 : \texttt{token}, u_2 : \texttt{token}, u_3 : \texttt{token}, c_0 : \texttt{count z}$$

$\boxed{\Gamma;\Delta \vdash O \Rightarrow A^-}$ Typing Atomic Objects

$$\frac{x : A^- \in \Gamma}{\Gamma;\Delta \vdash x \Rightarrow A^-}\ var \qquad \frac{c : A^- \in \Sigma}{\Gamma;\Delta \vdash c \Rightarrow A^-}\ const \qquad \frac{}{\Gamma; u : A^- \vdash u \Rightarrow A^-}\ lvar$$

$$\frac{\Gamma;\Delta_1 \vdash O \Rightarrow \Pi p : A^+.B^- \quad \Gamma;\Delta_2 \vdash M \Leftarrow A^+}{\Gamma;\Delta_1,\Delta_2 \vdash O\ M \Rightarrow [M/p]B^-}\ app$$

$$\frac{\Gamma;\Delta \vdash O \Rightarrow A_1^- \& A_2^-}{\Gamma;\Delta \vdash \pi_1 O \Rightarrow A_1^-}\ proj_1 \qquad \frac{\Gamma;\Delta \vdash O \Rightarrow A_1^- \& A_2^-}{\Gamma;\Delta \vdash \pi_2 O \Rightarrow A_2^-}\ proj_2$$

$\boxed{\Gamma;\Delta \vdash N \Leftarrow A^-}$ Typing Normal Objects

$$\frac{\Gamma;\Delta; p : A^+ \vdash \Gamma';\Delta' \qquad \Gamma';\Delta' \vdash N \Leftarrow A^-}{\Gamma;\Delta \vdash \lambda p.N \Leftarrow \Pi p : A^+.A^-}\ \Pi I \qquad \frac{\Gamma;\Delta \vdash N_1 \Leftarrow A^- \qquad \Gamma;\Delta \vdash N_2 \Leftarrow B^-}{\Gamma;\Delta \vdash \langle N_1; N_2\rangle \Leftarrow A^- \& B^-}\ \& I$$

$$\frac{\Gamma;\Delta \vdash E \Leftarrow \Gamma';\Delta' \quad \Gamma';\Delta' \vdash M \Leftarrow A^+}{\Gamma;\Delta \vdash \text{let } E \text{ in } \{M\} \Leftarrow \{A^+\}}\ \{\}I \qquad \frac{\Gamma;\Delta \vdash O \Rightarrow P}{\Gamma;\Delta \vdash O \Leftarrow P}\ blur$$

$\boxed{\Gamma;\Delta \vdash M \Leftarrow A^+}$ Typing Monadic Objects

$$\frac{\Gamma;\Delta_1 \vdash M_1 \Leftarrow A_1^+ \qquad \Gamma;\Delta_2 \vdash M_2 \Leftarrow A_2^+[M_1/p]}{\Gamma;\Delta_1,\Delta_2 \vdash \langle\langle M_1, M_2\rangle\rangle \Leftarrow \exists p : A_1^+.A_2^+}\ \exists I \qquad \frac{}{\Gamma;\cdot \vdash 1 \Leftarrow 1}\ 1I$$

$$\frac{\Gamma;\Delta \vdash N \Leftarrow A^-}{\Gamma;\Delta \vdash\ \downarrow N \Leftarrow \downarrow A^-}\ \downarrow I \qquad \frac{\Gamma;\cdot \vdash N \Leftarrow A^-}{\Gamma;\cdot \vdash\ !N \Leftarrow\ !A^-}\ \downarrow I$$

$\boxed{\Gamma;\Delta \vdash E \Leftarrow \Gamma';\Delta'}$ Typing Trace Objects

$$\frac{}{\Gamma;\Delta \vdash \cdot \Leftarrow \Gamma;\Delta}\ empty$$

$$\frac{\Gamma;\Delta_1 \vdash O \Leftarrow A^- \quad \Gamma;\Delta_2; p : A^- \vdash \Gamma';\Delta' \quad \Gamma';\Delta' \vdash E \Leftarrow \Gamma'';\Delta''}{\Gamma;\Delta_1,\Delta_2 \vdash p = O; E \Leftarrow \Gamma'';\Delta''}\ cons$$

Figure 4: Static Semantics

we apply `rule` to u_2 and c_1, resulting in contexts

$$\cdot;\cdot, u_1 : \texttt{token}, u_3 : \texttt{token}, c_1 : \texttt{count (s z)}.$$

As a second step, we apply `rule` to u_1 and c_1 and obtain

$$\cdot;\cdot, u_3 : \texttt{token}, c_2 : \texttt{count (s (s z))}.$$

And finally, we apply the same `rule` one more time to u_3 and c_2 and obtain

$$\cdot;\cdot, c_3 : \texttt{count (s (s (s z)))}$$

as the final context pair $\Gamma_3;\Delta_3$. Putting all pieces together we obtain that the trace is well-typed:

$$\Gamma_0;\Delta_0 \vdash\ \downarrow c_1 = \texttt{rule}\ u_2\ c_0;\ \downarrow c_2 = \texttt{rule}\ u_1\ c_1;\ \downarrow c_3 = \texttt{rule}\ u_3\ c_2; \cdot \Leftarrow \Gamma_3;\Delta_3$$

3.3 Definitional Equivalences

Our framework provides the standard equivalences α and β including and strict definition expansions. In addition, for trace objects, our framework also provides a notion of equivalence that is based on reordering the individual steps unless such a reordering would break dependencies. Concretely, two trace objects $E_1 = (p_1 = O_1; p_2 = O_2; E_1')$ and $E_2 = (p_2 = O_2; p_1 = O_1; E_2')$ are equivalent, written as $E_1 \equiv E_2$ iff O_2 does not refer to p_1 and O_1 does not refer to p_2 and $E_2' \equiv E_1'$. The justification for this definition is that traces correspond to cuts modulo commutative conversions. If we were to extend the example above and allowed two counters that count in parallel, the different trace objects would be equivalent modulo interleavings of the two traces.

4 Encoding Processes in Celf

We now move to describe a concrete Celf implementation of the two-layer system presented in Section 2.1 and argue its adequacy.

4.1 Typed Processes

Encoding of the formulas and typing rules is presented below. We use `o` to stand for a type of a session. Two type families, `hyp : o → type` and `conc : o → type`, describe, respectively, typed variables on the left of a sequent and process terms of a certain type. This kind of encoding is standard for representing sequent calculus in the LF methodology.

```
% Types and Judgments
o    : type.
conc : o → type.
hyp  : o → type.

% Formulas
one   : o.
tens  : o → o → o.
lolli : o → o → o.
with  : o → o → o.
plus  : o → o → o.
```

Unsurprisingly, the encoding of types can be formally described as a function:

$$\boxed{\ulcorner A \urcorner = \texttt{A}}$$

$$\begin{aligned}
\ulcorner 1 \urcorner &= \texttt{one} \\
\ulcorner A \otimes B \urcorner &= \texttt{tens}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\
\ulcorner A \multimap B \urcorner &= \texttt{lolli}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\
\ulcorner A \,\&\, B \urcorner &= \texttt{with}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\
\ulcorner A \oplus B \urcorner &= \texttt{plus}\ \ulcorner A \urcorner\ \ulcorner B \urcorner
\end{aligned}$$

The encoding of a context Δ is equally straightforward: $\ulcorner \Delta \urcorner$ introduces a linear assumption $\texttt{u} \mathbin{\hat{:}} \texttt{hyp}\ \ulcorner A \urcorner$ for each $u : A \in \Delta$.

$$\boxed{\ulcorner \Delta \urcorner = \texttt{D}}$$

$$\begin{aligned}
\ulcorner . \urcorner &= . \\
\ulcorner \Delta, x : A \urcorner &= \ulcorner \Delta \urcorner, \texttt{x} \mathbin{\hat{:}} \texttt{hyp}\ \ulcorner A \urcorner
\end{aligned}$$

Having type families `hyp` and `conc` corresponding, respectively, to the left and right side of the sequent, we continue by encoding the typing rules for simple processes presented on Figure 2. We use the $\multimap$ operator built into Celf to ensure that the assumptions are indeed treated linearly. As usual, we make use of the Higher-Order Abstract Syntax (HOAS) to describe hypothetical judgments as linear functions (see e.g. `lolliR` case).

```
% Typing
ax  : hyp A ⊸ conc A.
cut : conc A ⊸ (hyp A ⊸ conc C) ⊸ conc C.

% The Multiplicative Fragment
oneR   : conc one.
oneL   : conc C ⊸ (hyp one ⊸ conc C).
tensR  : conc A ⊸ conc B ⊸ conc (tens A B).
tensL  : (hyp A ⊸ hyp B ⊸ conc C) ⊸ (hyp (tens A B) ⊸ conc C).
lolliR : (hyp A ⊸ conc B) ⊸ (conc (lolli A B)).
lolliL : conc A ⊸ (hyp B ⊸ conc C) ⊸ (hyp (lolli A B) ⊸ conc C).

% The Additive Fragment
withR  : conc A & conc B ⊸ conc (with A B).
withL1 : (hyp A ⊸ conc C) ⊸ (hyp (with A B) ⊸ conc C).
withL2 : (hyp B ⊸ conc C) ⊸ (hyp (with A B) ⊸ conc C).
plusR1 : conc A ⊸ conc (plus A B).
plusR2 : conc B ⊸ conc (plus A B).
plusL  : (hyp A ⊸ conc C) & (hyp B ⊸ conc C) ⊸ (hyp (plus A B) ⊸ conc C).
```

The above rules are the representation of well-typed processes $P \rhd \Delta \vdash A$. The encoding of well-typed processes is therefore given as:

$\boxed{\ulcorner P \urcorner = \texttt{M}}$

$$
\begin{array}{lcl}
\ulcorner \mathbf{fwd}_{\odot}\ x \urcorner & = & \texttt{ax x} \\
\ulcorner \nu x^{A}.(P \mid Q) \urcorner & = & \texttt{cut } \ulcorner P \urcorner\ (\widehat{\lambda}\ \texttt{x}\ .\ \ulcorner Q \urcorner) \\
\ulcorner \mathbf{end}_{\odot} \urcorner & = & \texttt{oneR} \\
\ulcorner \mathbf{wait}^{x}.P \urcorner & = & \texttt{oneL } \ulcorner P \urcorner\ \texttt{x} \\
\ulcorner \nu y^{A}.\odot\langle y \rangle.(P \mid Q) \urcorner & = & \texttt{tensR } \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner x(y).P \urcorner & = & \texttt{tensL } (\widehat{\lambda}\texttt{y}\ .\ \widehat{\lambda}\texttt{x}\ .\ \ulcorner P \urcorner)\ \texttt{x} \\
\ulcorner \odot(y).P \urcorner & = & \texttt{lolliR } (\widehat{\lambda}\texttt{y}\ .\ \ulcorner P \urcorner) \\
\ulcorner \nu y^{A}.x\langle y \rangle.(P \mid Q) \urcorner & = & \texttt{lolliL } \ulcorner P \urcorner\ (\widehat{\lambda}\ \texttt{x}\ .\ \ulcorner Q \urcorner)\ \texttt{x} \\
\ulcorner \odot.\mathbf{case}\ (P, Q) \urcorner & = & \texttt{withR } \langle \ulcorner P \urcorner\ ,\ \ulcorner Q \urcorner \rangle \\
\ulcorner x.\mathbf{case}\ (P, Q) \urcorner & = & \texttt{plusL } \langle \widehat{\lambda}\texttt{x}\ .\ \ulcorner P \urcorner,\ \widehat{\lambda}\texttt{x}.\ \ulcorner Q \urcorner\ \rangle\ \texttt{x} \\
\ulcorner x.\mathbf{inl}; P \urcorner & = & \texttt{withL1 } (\widehat{\lambda}\texttt{x}\ .\ \ulcorner P \urcorner)\ \texttt{x} \\
\ulcorner x.\mathbf{inr}; Q \urcorner & = & \texttt{withL2 } (\widehat{\lambda}\texttt{x}\ .\ \ulcorner Q \urcorner)\ \texttt{x} \\
\ulcorner \odot.\mathbf{inl}; P \urcorner & = & \texttt{plusR1 } \ulcorner P \urcorner \\
\ulcorner \odot.\mathbf{inr}; Q \urcorner & = & \texttt{plusR2 } \ulcorner Q \urcorner
\end{array}
$$

To argue adequacy, we show how the encoding gives well-typed derivations. We present only

some of the translations, the others are obtained in a similar manner.

$$\boxed{P \rhd \Delta \vdash A} \qquad \boxed{\ulcorner\Delta\urcorner \vdash \ulcorner P\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner A\urcorner}$$

$$\frac{}{\mathbf{fwd}_{\odot}\ x \rhd x : C \vdash C} \qquad \frac{}{\mathtt{x} \mathbin{\hat{:}} \mathtt{hyp}\ \ulcorner C\urcorner \vdash \mathtt{ax}\ \hat{}\ \mathtt{x} : \mathtt{conc}\ \ulcorner C\urcorner}$$

$$\frac{P \rhd \Gamma \vdash A \qquad Q \rhd \Delta, x : A \vdash C}{\nu x^{A}.(P \mid Q) \rhd \Gamma, \Delta \vdash C} \qquad \frac{\ulcorner\Gamma\urcorner \vdash \ulcorner P\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner A\urcorner \qquad \dfrac{\ulcorner\Delta\urcorner, \mathtt{x} \mathbin{\hat{:}} \mathtt{hyp}\ \ulcorner A\urcorner \vdash \ulcorner Q\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner C\urcorner}{\ulcorner\Delta\urcorner \vdash \widehat{\lambda}\ \mathtt{x}\ .\ \ulcorner Q\urcorner \mathbin{:} \mathtt{hyp}\ \ulcorner A\urcorner \multimap \mathtt{conc}\ \ulcorner C\urcorner}}{\ulcorner\Gamma\urcorner\ \ulcorner\Delta\urcorner \vdash \mathtt{cut}\ \hat{}\ \ulcorner P\urcorner\ \hat{}\ (\widehat{\lambda}\ \mathtt{x}\ .\ \ulcorner Q\urcorner) \mathbin{:} \mathtt{conc}\ \ulcorner C\urcorner}$$

$$\frac{}{\mathbf{end}_{\odot} \rhd \vdash 1} \qquad \frac{}{\mathtt{.} \vdash \mathtt{oneR} \mathbin{:} \mathtt{conc\ one}}$$

$$\frac{P \rhd \Delta \vdash C}{\mathbf{wait}^{x}.P \rhd \Delta, x : 1 \vdash C} \qquad \frac{\dfrac{\ulcorner\Delta\urcorner \vdash \ulcorner P\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner C\urcorner}{\ulcorner\Delta\urcorner \vdash \mathtt{oneL}\ \ulcorner P\urcorner \mathbin{:} \mathtt{hyp\ one} \multimap \mathtt{conc}\ \ulcorner C\urcorner} \qquad \frac{}{\mathtt{x} \mathbin{\hat{:}} \mathtt{hyp\ one} \vdash \mathtt{x} \mathbin{:} \mathtt{hyp\ one}}}{\ulcorner\Delta\urcorner, \mathtt{x} \mathbin{\hat{:}} \mathtt{hyp\ one} \vdash \mathtt{oneL}\ \ulcorner P\urcorner\ \mathtt{x} \mathbin{:} \mathtt{conc}\ \ulcorner C\urcorner}$$

$$\frac{P \rhd \Gamma \vdash A \quad Q \rhd \Delta \vdash B}{\nu y^{A}.\odot\langle y\rangle.(P \mid Q) \rhd \Gamma, \Delta \vdash A \otimes B} \qquad \frac{\ulcorner\Gamma\urcorner \vdash \ulcorner P\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner A\urcorner \qquad \ulcorner\Delta\urcorner \vdash \ulcorner Q\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner B\urcorner}{\ulcorner\Gamma\urcorner, \ulcorner\Delta\urcorner \vdash \mathtt{tensR}\ \ulcorner P\urcorner\ \ulcorner Q\urcorner \mathbin{:} \mathtt{conc}\ (\mathtt{tens}\ \ulcorner A\urcorner\ \ulcorner B\urcorner)}$$

$$\frac{P \rhd \Delta, y : A, x : B \vdash C}{x(y).P \rhd \Delta, x : A \otimes B \vdash C} \qquad \frac{\dfrac{\dfrac{\ulcorner\Delta\urcorner, \mathtt{y} \mathbin{\hat{:}} \mathtt{hyp}\ \ulcorner A\urcorner, \mathtt{x} \mathbin{\hat{:}} \mathtt{hyp}\ \ulcorner B\urcorner \vdash \ulcorner P\urcorner \mathbin{:} \mathtt{conc}\ \ulcorner C\urcorner}{\ulcorner\Delta\urcorner, \mathtt{y} \mathbin{\hat{:}} \mathtt{hyp}\ \ulcorner A\urcorner \vdash \widehat{\lambda}\mathtt{x}\ .\ \ulcorner P\urcorner \mathbin{:} \mathtt{hyp}\ \ulcorner B\urcorner \multimap \mathtt{conc}\ \ulcorner C\urcorner} \qquad \frac{}{\mathtt{x} \mathbin{\hat{:}} \mathtt{hyp}\ \ldots \vdash \mathtt{x} \mathbin{:} \mathtt{hyp}\ \ldots}}{\ulcorner\Delta\urcorner \vdash \widehat{\lambda}\mathtt{y}\ .\ \widehat{\lambda}\ \mathtt{x}\ .\ \ulcorner P\urcorner \mathbin{:} \mathtt{hyp}\ \ulcorner A\urcorner \multimap \mathtt{hyp}\ \ulcorner B\urcorner \multimap \mathtt{conc}\ \ulcorner C\urcorner}}{\ulcorner\Delta\urcorner, \mathtt{x} \mathbin{\hat{:}} \mathtt{hyp}\ (\mathtt{tens}\ \ulcorner A\urcorner\ \ulcorner B\urcorner) \vdash \mathtt{tensL}\ (\lambda\mathtt{y}\hat{}\ .\ \widehat{\lambda}\mathtt{x}\ .\ \ulcorner P\urcorner)\ \mathtt{x} \mathbin{:} \mathtt{conc}\ \ulcorner C\urcorner}$$

4.2 Extended Processes and Reduction Rules

Besides simple typed processes, the system presented in Section 2.1 makes use of extended processes. These lists of let-bindings associate every process with a channel on which it provides a behaviour described in its type. In the Celf encoding, we will represent a single let binding using a `proc` type family:

```
proc :  conc A → hyp A → type.
```

An LF-object of type `conc A` is a process realising behaviour of `A`, while an object of type `hyp A` can only be an assumption from the context.

Looking at the congruences described for these let bindings, it is meaningful to skip the explicit encoding of lists of `proc`s, and instead describe $\ulcorner E\urcorner$ as a (mixed, linear-intuitionistic) context of `proc C H` assumptions. Notice that here elements of type `hyp A` in the context can no longer be linear, as we use them as indices to type family `proc`. Indeed, $\ulcorner E\urcorner$ is a Celf context `G` of schema **block** `{h : hyp A, p ̂: proc P h}`. In other words, an extended process is encoded as:

$$\boxed{\ulcorner E\urcorner = \mathtt{G}}$$

$$\begin{aligned}
\ulcorner P\urcorner^{z} &= \mathtt{z} \mathbin{:} \mathtt{hyp\ 1},\ \mathtt{p} \mathbin{\hat{:}} \mathtt{proc}\ \ulcorner P\urcorner\ \mathtt{z} \\
\ulcorner \mathbf{let}\ x := P\ \mathbf{in}\ E\urcorner^{z} &= \mathtt{x} \mathbin{:} \mathtt{hyp\ A},\ \mathtt{p} \mathbin{\hat{:}} \mathtt{proc}\ \ulcorner P\urcorner\ \mathtt{x},\ \ulcorner E\urcorner^{z} \qquad \text{where } P \text{ is of type } A
\end{aligned}$$

Next we present the operational semantics in our encoding. As we are interested only in capturing reductions reconstructed on extended processes, we do not allow reductions directly in simple typed processes. In our encoding simple processes are static, and it is in the extended process environment E that things actually communicate. For instance, the cut reduction rule simply pushes cut up: we do not reduce in a simple process `(cut P (λx. Q x))` but instead spawn two new let-bindings:

```
red/cut : proc (cut P (λx. Q x)) C ⊸
          { Exists a.  proc P a ⊗
```

```
          proc (Q a) C
        }.
```

Simple let-bindings using axioms can be safely removed:

```
red/ax/left: proc D H ⊸
             proc (ax H) C ⊸
             { proc D C }.

red/ax/right: proc (ax H) C ⊸
              proc (Q C) C' ⊸
              { proc (Q H) C' }.
```

We think about these rules as defining a multiset rewrite system. Reading the rules above directly as context transformations gives the following context transformations.

`r : G ↦ G'` Rule `r` describes a transformation between mixed contexts `G` and `G'`

```
red/cut      : (G, q ⍰ proc (cut P (λx . Q x)) C ) ↦
                          (G, a : hyp _, p_1 ⍰ proc P a, p_2 ⍰ proc (Q a) C )
red/ax/left  : (G, q_1 ⍰ proc D H, q_2 ⍰ proc (ax H) C ) ↦  (G, p ⍰ proc D C )
red/ax/right : (G, q_1 ⍰ proc (ax H) C, q_2 ⍰ proc (Q C) C' ) ↦
                                                    (G, p ⍰ proc (Q H) C' )
```

The rest of the reduction rules' encodings are given below. We give them only as Celf code, as it is easy to imagine what the context-rewriting rules will look like.

```
% Multiplicative fragment
red/one : proc oneR C ⊸
          proc (oneL D C) C' ⊸
          { proc D C' }.

red/comm : proc (tensR P1 P2) H ⊸
           proc (tensL (λu1. λu2. Q u1 u2) H) C'' ⊸
           { Exists a. proc P1 a ⊗
             Exists b. proc P2 b ⊗
             proc (Q a b) C''
           }.

red/lolli : proc (lolliR (λu. P u)) C ⊸
            proc (lolliL Q1 (λv. Q2 v) C) C'' ⊸
            { Exists a. proc Q1 a ⊗
              Exists b. proc (P a) b ⊗
              proc (Q2 b) C''
            }.

% Additive fragment
red/with1 : proc (withR ⟨ P1 , P2 ⟩) C ⊸
            proc (withL1 (λv. Q v) C) C'' ⊸
            { Exists a. proc P1 a ⊗
              proc (Q a) C''
            }.

red/with2 : proc (withR ⟨ P1 , P2 ⟩) C ⊸
            proc (withL2 (λv. Q v) C) C'' ⊸
            { Exists a. proc P2 a ⊗
              proc (Q a) C''
            }.

red/plus1 : proc (plusR1 P) C ⊸
```

```
            proc (plusL ⟨ (λv. Q1 v), (λv. Q2 v) ⟩ C) C'' ⊸
           { Exists a. proc P a ⊗
             proc (Q1 a) C''
           }.

red/plus2 : proc (plusR2 P) C ⊸
            proc (plusL ⟨ (λv. Q1 v), (λv. Q2 v) ⟩ C) C'' ⊸
           { Exists a. proc P a ⊗
             proc (Q2 a) C''
           }.
```

The observant reader will notice that we only have the principal cut rules. Viewed as a process calculus, the congruence rules are missing, and viewed as cut elminiation, the commuting cut rules are missing. The `red/cut` rule lifts the cut from the term into the context, and that makes the two processes of the cut independent. When a reduction rule is applied, the relevant processes are selected from the context without the need to care about their spatial location.

Furthermore, because of the definitional equality of CLF, which is described in Section 3.3, we have that traces that differ only in the order of independent reduction steps are considered equal.

4.3 Adequacy of the Encoding

As already discussed in Section 3, adequacy of the encodings for a concurrent system has to be established between equivalence classes of derivations. Giving complete proofs of adequacy is beyond the scope of this paper, instead we focus here on providing informal intuitions.

The rationale for adequacy of typing rules has already been partially sketched when the encoding was introduced. It does not use the concurrency monad, instead it stays within the LLF fragment of the system, which has canonical forms. Therefore in this fragment we do not need to concern ourselves with commuting conversions and equivalence classes. Since the correspondence between the typing rules of Figure 2 and their Celf encoding is tight, providing an inverse of $\ulcorner J \urcorner$ is purely mechanical.

Theorem 1 (Adequacy: Processes)**.**
All of the following are compositional bijections:

1. $\ulcorner A \urcorner$ *where* A *is a well-formed type;*
2. $\ulcorner \Delta \urcorner$ *where* Δ *is a well-formed context;*
3. $\ulcorner P \rhd \Delta \vdash A \urcorner = \ulcorner \Delta \urcorner \vdash \ulcorner P \urcorner : \ulcorner A \urcorner$ *where* $\mathcal{D} :: P \rhd \Delta \vdash A$ *is a correct typing derivation according to the rules of Figure 2.*

For extended processes, the adequacy is also easy to see. An extended process E is in fact a list of let bindings which can be reordered using the commuting conversion, provided the well-typedness of E is maintained. In the encoding $\ulcorner E \urcorner$ we have chosen for these extended processes, we use context as an abstraction for these lists. Here as well the reordering is possible if and only if the context remains well-formed after reordering.

$$\boxed{\Gamma \vdash E} \qquad \boxed{\ulcorner \Gamma \urcorner,\ \ulcorner E \urcorner^{y} \vdash \texttt{y : hyp one}}$$

$$\frac{P \rhd \Gamma \vdash 1}{\Gamma \vdash P} \qquad \frac{\ulcorner \Gamma \urcorner \vdash \ulcorner \texttt{P} \urcorner \texttt{ : conc one}}{\ulcorner \Gamma \urcorner,\ \texttt{y: hyp one},\ \texttt{p} \mathbin{\hat{:}} \texttt{proc } \ulcorner \texttt{P} \urcorner\ \texttt{y} \vdash \texttt{y : hyp one}}$$

$$\frac{P \rhd \Gamma_1 \vdash A \quad \Gamma_2, x : A \vdash E}{\Gamma_1, \Gamma_2 \vdash \textbf{let}\ x := P\ \textbf{in}\ E} \qquad \frac{\ulcorner \Gamma_1 \urcorner \vdash \ulcorner P \urcorner \texttt{ : conc } \ulcorner A \urcorner \quad \ulcorner \Gamma_2 \urcorner,\ \texttt{x : hyp } \ulcorner A \urcorner,\ \ulcorner E \urcorner^{z} \vdash \texttt{ctx}}{\ulcorner \Gamma_1 \urcorner,\ \ulcorner \Gamma_2 \urcorner,\ \texttt{x : hyp A},\ \texttt{p} \mathbin{\hat{:}} \texttt{proc } \ulcorner P \urcorner\ \texttt{x},\ \ulcorner E \urcorner^{z} \vdash \texttt{ctx}}$$

Theorem 2 (Adequacy: Extended Processes)**.**
An encoding function $\ulcorner \Gamma \vdash E \urcorner$ *is a compositional bijection. More precisely, there is a compositional bijection between*

1. *well-typed extended processes* $\Gamma \vdash E$,
2. *and those* $\texttt{G, D} \vdash y_k \texttt{ : hyp one}$ *where*
   - *i)* $\texttt{G}$ *is a context of the form* $x_0 \texttt{:hyp } A_0 \texttt{, } \ldots$,
   - *ii)* $\texttt{D}$ *is a context of the form* $y_0 \texttt{:hyp } B_0 \texttt{, } p_0 \hat{:} \texttt{proc } P_0 \ y_0 \texttt{, } \ldots$,
   - *iii) the variable* y_i *does not occur in* P_i,
   - *iv) the* $\texttt{hyp}$*-variables are used linearly by the processes* P_j,
   - *v) except the variable* y_k *that is not used by any process.*

We remark, first, that G is empty if the extended process is closed and, second, that the side conditions iii) – v) are closely related to generative invariants [Sim12].

The adequacy of the encoding we give for reduction rules has to be carefully stated. First, it is crucial to note that we do not aim at encoding all the possible process reductions, only those that preserve well-typedness. Second, as we describe reductions directly on the extended processes, a single context rewrite step in our Celf encoding corresponds to multiple steps of extended process rewrites described in Figure 1.

To be more precise, instead of action tracing that is preserved as we traverse the list of process bindings, we directly expose both parts of the communication when we describe the reduction and rely on context equivalence to take care of any possible reorderings. Exposing two communicating parties at once is consistent with the first observation, as we are interested only in faithful representation of τ actions. Further, we make sure no communication will happen within a simple process by lifting any communication (cut) to the extended process level.

Finally, we present an adequacy result for reductions. Informally it says that all interleavings of extended process reductions are considered equivalent inside the framework.

Theorem 3 (Adequacy: Reductions)**.**
If E *is a well-typed extended process* $\Gamma \vdash E : \Delta$ *and there is a reduction* $E \xrightarrow{\tau} F$*, then there is a unique (modulo reordering in* E*) term describing application of context rewriting rules in order to transform* $\ulcorner \Gamma \urcorner$; $\ulcorner E \urcorner$ *into* $\ulcorner \Gamma \urcorner$; $\ulcorner F \urcorner$.
Conversely if $\texttt{r}$ *is a trace of context rewrites (each context pair satisfying similar invariants as in Theorem 2) then* $\texttt{r : G; D} \mapsto \texttt{G'; D'}$ *then there exists a unique pair of extended processes* E *and* F *such that* $E \xrightarrow{\tau} F$*. and* $\ulcorner E \urcorner = \texttt{D}$, $\ulcorner F \urcorner = \texttt{D'}$.

5 Focusing

Focusing was introduced by Andreoli [And92] as a way to reduce the search space of proof search in classical linear logic. His observation is that "some proofs are the same up to some irrelevant reordering or simplification of inference rules".

The most obvious example of that is the order in which *invertible* rules are applied. A rule

$$\frac{\mathcal{J}_0 \quad \ldots \quad \mathcal{J}_n}{\mathcal{J}}$$

is considered invertible if the rules

$$\frac{\mathcal{J}}{\mathcal{J}_0} \qquad \ldots \qquad \frac{\mathcal{J}}{\mathcal{J}_n}$$

are admissible. An informal reading is that applying invertible rules cannot go wrong – at least when it comes to provability.

The second observation is that non-invertible rules can be chained together in a restricted fashion. Whenever a non-invertible rule is applied, we can continue to only consider further non-invertible rules applied to the principal formula's subformulas.

In linear logic, we can divide the connectives into two disjoint groups. The *positive* connectives consist of $\otimes$, 1 and $\oplus$, while the *negative* connectives include $\multimap$ and $\&$. All the positive connectives have invertible left rules, and non-invertible right rules. Conversely, all the negative connectives have non-invertible left rules, and invertible right rules.

Combining these ideas, we can consider proofs alternating between two phases: the inversion phase –where invertible rules are applied– and the focus phase –where non-invertible rules are applied. The logical content of these two phases is different like night and day: the inversion phase contains essentially no information, while the focus phase is packed with information. Another way to look at this is that the inversion phase corresponds to clerical work, while the focus phase is experts work [Chi15]. However we choose to think about these two phases, they are completely separate, with explicit rules allowing to move from one phase to the other.

A final property which is often considered important is the possibility to only consider proofs where the phases are *maximal* while maintaining provability. The possible shapes of a maximally focused proof are dictated by the polarity of the formulas. This means that we get a strong notion of normal forms for proofs.

Focusing and Sessions. We want to investigate this notion of normal form in the context of process calculi. The presence of normal forms can be exploited when proving properties about processes – given that the property respects normal forms.

As it turns out, focusing will have a very pleasing meaning in process calculi. All the invertible rules correspond to type checking receiving operations, while all the non-invertible rules correspond to sending operations. This is, in some, sense quite natural: receiving information is a clerical work, while sending is an expert's work, as a decision must be made about what is sent. The two phases of focused sessions are thus interpreted as the sending phase and the receiving phase.

5.1 Extending Processes to Focused Processes

We continue with the same types as before:

$$A, B ::= a \mid 1 \mid A \otimes B \mid A \multimap B \mid A \,\&B \mid A \oplus B$$

In the same way that proof search alternates between inversion and focusing, processes alternate between sending and receiving. We have two variants of the sending phase, depending on whether the sending occurs on a channel in the context or on the implicit "goal" channel. This gives rise to the following three judgments:

- $P \rhd \Gamma \vdash A$: The process P receives on Γ and A.
- $S \rhd \Gamma\ [A] \vdash B$: The process S is sending on a channel with session type A.
- $V \rhd \Gamma \vdash [A]$: The process V is sending on a channel with session type A.

Corresponding to these judgments are three different kinds of processes:

$$\begin{aligned} P, Q &::= x(y).P \mid \mathbf{wait}^x.P \mid x.\mathbf{case}(P,\, Q) \mid (x).P \mid \mathbf{case}(P,\, Q) \\ &\quad \mid \mathbf{on}\ \ x\ \mathbf{do}\ V \mid \mathbf{do}\ \ S \mid \nu x^A.(P \mid Q) \\ S, T &::= \mathbf{fwd}\ x \mid \langle S \rangle T \mid \mathbf{end} \mid \mathbf{inl}; S \mid \mathbf{inr}; S \mid \mathbf{blur}\ P \\ V, W &::= \mathbf{fwd} \mid \langle S \rangle V \mid \mathbf{inl}; V \mid \mathbf{inr}; V \mid \mathbf{blur}\ x.P \end{aligned}$$

The receiving processes $\mathbf{on}\ \ x\ \mathbf{do}\ V$ and $\mathbf{do}\ \ S$ mark the transition from a receiving phase to a sending phase. $\mathbf{on}\ \ x\ \mathbf{do}\ V$ starts to send according to the process V on the channel x, while $\mathbf{do}\ \ S$ starts sending according to the process S on the return channel. In the same fashion, the sending processes $\mathbf{blur}\ x.P$ ends a left-sending phase, and will continue as the receiving process P with the channel it was sending on named x, and $\mathbf{blur}\ P$ will end a sending phase on the implicit channel and enter a receiving phase.

Typing of these process terms is defined in Figure 5. Most of the typing rules are as before, but the change of judgments imposes a specific structure on the processes. Interestingly, the single axiom rule in the unfocused system emerges as two rules: axiom on the left and axiom on the right. There are also four new rules governing the transition between phases: the two focus rules select a channel to start sending on, while the two blur rules terminate a sending phase and return to a receiving phase.

$$\frac{}{\mathbf{fwd} \rhd\ [A] \vdash A}\ ax_L \qquad \frac{}{\mathbf{fwd}\ x \rhd x : A \vdash [A]}\ ax_R$$

$$\frac{}{\mathbf{end} \rhd \cdot \vdash [1]}\ 1_R \qquad \frac{P \rhd \Gamma \vdash C}{\mathbf{wait}^x.P \rhd \Gamma, x : 1 \vdash C}\ 1_L$$

$$\frac{S \rhd \Gamma \vdash [A] \quad T \rhd \Delta \vdash [B]}{\langle S \rangle T \rhd \Gamma, \Delta \vdash [A \otimes B]}\ \otimes_R \qquad \frac{P \rhd \Gamma,\, y : A,\, x : B \vdash C}{x(y).P \rhd \Gamma,\, x : A \otimes B \vdash C}\ \otimes_L$$

$$\frac{P \rhd \Gamma, x : A \vdash B}{(x).P \rhd \Gamma \vdash A \multimap B}\ \multimap_R \qquad \frac{S \rhd \Gamma \vdash [A] \quad V \rhd \Delta\ [B] \vdash C}{\langle S \rangle V \rhd \Gamma, \Delta\ [A \multimap B] \vdash C}\ \multimap_L$$

$$\frac{P \rhd \Gamma \vdash A \quad Q \rhd \Gamma \vdash B}{\mathbf{case}(P,\, Q) \rhd \Gamma \vdash A \& B}\ \&_R \qquad \frac{P \rhd \Gamma, x : A \vdash C \quad Q \rhd \Gamma, x : B \vdash C}{x.\mathbf{case}(P,\, Q) \rhd \Gamma, x : A \oplus B \vdash C}\ \oplus_L$$

$$\frac{S \rhd \Gamma \vdash [A_1]}{\mathbf{inl}; S \rhd \Gamma \vdash [A_1 \oplus A_2]}\ \oplus_{R_1} \qquad \frac{S \rhd \Gamma \vdash [A_2]}{\mathbf{inl}; S \rhd \Gamma \vdash [A_1 \oplus A_2]}\ \oplus_{R_2}$$

$$\frac{V \rhd \Gamma\ [A_2] \vdash C}{\mathbf{inl}; V \rhd \Gamma\ [A_1 \& A_2] \vdash C}\ \&_{L_1} \qquad \frac{V \rhd \Gamma\ [A_2] \vdash C}{\mathbf{inr}; V \rhd \Gamma\ [A_1 \& A_2] \vdash C}\ \&_{L_2}$$

$$\frac{V \rhd \Gamma\ [A] \vdash C}{\mathbf{on}\ \ x\ \mathbf{do}\ V \rhd \Gamma,\, x : A \vdash C}\ focus_L \qquad \frac{S \rhd \Gamma \vdash [A]}{\mathbf{do}\ \ S \rhd \Gamma \vdash A}\ focus_R$$

$$\frac{P \rhd \Gamma,\, x : A \vdash C}{\mathbf{blur}\ x.P \rhd \Gamma\ [A] \vdash C}\ blur_L \qquad \frac{P \rhd \Gamma \vdash C}{\mathbf{blur}\ P \rhd \Gamma \vdash [C]}\ blur_R$$

$$\frac{P \rhd \Gamma \vdash A \quad Q \rhd \Delta,\, x : A \vdash C}{\nu x^A.(P \mid Q) \rhd \Gamma,\, \Delta \vdash C}\ cut$$

Figure 5: Typing for Focused Processes

5.2 Adapting Operational Semantics

Extended processes are used to express *where* communication happens.

$$E ::= \mathbf{nil} \mid \mathbf{let}\ x := P\ \mathbf{in}\ E \mid \mathbf{let}\ x := V\ \mathbf{on}\ y\ \mathbf{in}\ E \mid \mathbf{let}\ x := S\ \mathbf{in}\ E$$

We consider extended processes equal up to exchange, the computation rules for extended processes are therefore the following:

$\mathbf{let}\ x := P\ \mathbf{in}\ \mathbf{let}\ y := \mathbf{fwd}\ \mathbf{on}\ x\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ y := P\ \mathbf{in}\ E$	axL
$\mathbf{let}\ x := P\ \mathbf{in}\ \mathbf{let}\ y := \mathbf{fwd}\ x\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ y := P\ \mathbf{in}\ E$	axR
$\mathbf{let}\ x := (\nu y.(P \mid Q))\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ y := P\ \mathbf{in}\ \mathbf{let}\ x := Q\ \mathbf{in}\ E$	cut
$\mathbf{let}\ x := \mathbf{end}\ \mathbf{in}\ \mathbf{let}\ y := \mathbf{wait}^x.P\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ y := P\ \mathbf{in}\ E$	one
$\mathbf{let}\ x := \langle S\rangle T\ \mathbf{in}\ \mathbf{let}\ z := x(y).P\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ y := S\ \mathbf{in}\ \mathbf{let}\ x := T\ \mathbf{in}\ \mathbf{let}\ z := P\ \mathbf{in}\ E$	comm
$\mathbf{let}\ x := \langle S\rangle V\ \mathbf{on}\ y\ \mathbf{in}\ \mathbf{let}\ y := (z).P\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ z := S\ \mathbf{in}\ \mathbf{let}\ x := V\ \mathbf{on}\ y\ \mathbf{in}\ \mathbf{let}\ y := P\ \mathbf{in}\ E$	lolli
$\mathbf{let}\ x := \mathbf{do}\ \ S\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ x := S\ \mathbf{on}\ E\ \mathbf{in}$	focusR
$\mathbf{let}\ x := \mathbf{on}\ \ y\ \mathbf{do}\ V\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ x := V\ \mathbf{on}\ y\ \mathbf{in}\ E$	focusL
$\mathbf{let}\ x := \mathbf{do}\ \ S\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ s := S\ \mathbf{in}\ E$	focusR
$\mathbf{let}\ x := \mathbf{blur}\ P\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ x := P\ \mathbf{in}\ E$	blurR
$\mathbf{let}\ x := \mathbf{blur}\ y.P\ \mathbf{on}\ y\ \mathbf{in}\ E$	$\mapsto$	$\mathbf{let}\ x := P\ \mathbf{in}\ E$	blurL

5.3 Encoding Focused Processes

The representation of this process calculus in Celf follows the same pattern as that of Section 4.

Encoding of well-typed processes. The three kind of processes are each represented by a type family. A receiving processes P such that $P \rhd \Gamma \vdash A$ is represented by the type class `conc A`. A sending process S such that $S \rhd \Gamma\ [A] \vdash B$ is represented by the type class `focL A B`. A sending process V such that $V \rhd \Gamma \vdash [A]$ is represented by the type class `focR A`.

A selection of the rules defining well-typed processes are:

```
axR     : hyp A —◦ focR A.
axL     : focL A A.
cut     : conc A —◦ (hyp A —◦ conc C) —◦ conc C.
focusL  : focL A B —◦ (hyp A —◦ conc B).
focusR  : focR A —◦ conc A.
blurL   : (hyp A —◦ conc B) —◦ focL A B.
blurR   : conc A —◦ focR A.
oneR    : focR one.
oneL    : conc C —◦ (hyp one —◦ conc C).
tensR   : focR A —◦ focR B —◦ focR (tens A B).
tensL   : (hyp A —◦ hyp B —◦ conc C) —◦ (hyp (tens A B) —◦ conc C).
lolliR  : (hyp A —◦ conc B) —◦ (conc (lolli A B)).
lolliL  : focR A —◦ focL B C —◦ (focL (lolli A B) C).
```

Encoding of well-typed extended processes. In addition to `proc P C` from the earlier section, we introduce two two new type families that represent sending processes.

```
proc  : conc A → hyp A → type.
procR : focR A → hyp A → type.
procL : hyp A → focL A C → hyp C → type.
```

The reading of `procL C S C'` is that the process `S` is currently interacting on channel `C`, but is defining the channel `C'`.

The rules describing the rewrite system is similar to earlier. Note that communication always involves two processes, and in the focused version, one process will always be in the receiving phase, while the other will be in one of the two possible sending phases.

6 Conclusion and Future Work

In this paper, we have developed a representation technique for processes in substructural operational semantics (SSOS) and the logical framework Celf. This technique provides concise and elegant representations of processes and their respective reduction semantics. Interleavings are directed supported by Celf and represented as equivalent trace objects.

Furthermore, we have applied our framework to two examples: a process algebra based on linear logic following closely [CP10], and a process algebra based on focused linear logic.

Closely related to our work is that of processes based polarized linear logic [PG15, PZ16]. Depending on its polarity, a channel can change directionality. In future work, we will will also represent this logic in Celf, and investigate the interactions between polarization and focusing.

We plan to also study global types, multi-party computations and co-inductive proofs in the focalized setting, building on prior work [CMSY15, CP16].

Acknowledgements: This publication was made possible by NPRP 7-988-1-178 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[Chi15] Zakaria Chihani. *Certification of First-order proofs in classical and intuitionistic logics*. PhD thesis, Ecole Polytechnique, 2015.

[CMS14] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *CONCUR*, pages 47–62, 2014.

[CMSY15] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In *CONCUR*, pages 412–426, 2015.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

[CP16] Luís Caires and Jorge Perez. Multiparty session types within a canonical binary theory, and beyond. In *FORTE*, 2016.

[CPPT13] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, pages 330–349, 2013.

[CPS+12] Iliano Cervesato, Frank Pfenning, Jorge Luis Sacchini, Carsten Schürmann, and Robert J. Simmons. Trace matching in a concurrent logical framework. In *Proceedings of the Seventh International Workshop on Logical Frameworks and Meta-languages, Theory and Practice*, LFMTP '12, pages 1–12, New York, NY, USA, 2012. ACM.

[HVK98] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, pages 22–138, 1998.

[LM16] Sam Lindley and Garrett Morris. Talking bananas: structural recursion for session types. In *ICFP*. ACM, 2016. To appear.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[PG15] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *Foundations of Software Science and Computation Structures*, pages 3–22. Springer, 2015.

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, pages 101–110, Washington, DC, USA, 2009. IEEE Computer Society.

[PZ16] Jennifer Paykin and Steve Zdancewic. Linear $\lambda\mu$ is CP (more or less). In *A List of Successes That Can Change The World*, pages 273–291, 2016.

[Sim12] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, 2012.

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

[Wad14] Philip Wadler. Propositions as sessions. *JFP*, 24(2–3):384–418, 2014. Also: ICFP, pages 273–286, 2012.

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2002. Revised May 2003.

A Appendix

Session Types in Celf

```
% Types and Judgments
o    : type.
conc : o → type.
hyp  : o → type.
proc :  conc A → hyp A → type.

% Formulas
one   : o.
tens  : o → o → o.
lolli : o → o → o.
with  : o → o → o.
plus  : o → o → o.

%%%%%%%%%
% Typing %
%%%%%%%%%

ax  : hyp A —◦ conc A.
cut : conc A —◦ (hyp A —◦ conc C) —◦ conc C.

% The Multiplicative Fragment
oneR   : conc one.
oneL   : conc C —◦ (hyp one —◦ conc C).
tensR  : conc A —◦ conc B —◦ conc (tens A B).
tensL  : (hyp A —◦ hyp B —◦ conc C) —◦ (hyp (tens A B) —◦ conc C).
lolliR : (hyp A —◦ conc B) —◦ (conc (lolli A B)).
lolliL : conc A —◦ (hyp B —◦ conc C) —◦ (hyp (lolli A B) —◦ conc C).
```

```
% The Additive Fragment
withR  : conc A & conc B ⊸ conc (with A B).
withL1 : (hyp A ⊸ conc C) ⊸ (hyp (with A B) ⊸ conc C).
withL2 : (hyp B ⊸ conc C) ⊸ (hyp (with A B) ⊸ conc C).
plusR1 : conc A ⊸ conc (plus A B).
plusR2 : conc B ⊸ conc (plus A B).
plusL  : (hyp A ⊸ conc C) & (hyp B ⊸ conc C) ⊸ (hyp (plus A B) ⊸ conc C).

%%%%%%%%%%%%%
% Reductions %
%%%%%%%%%%%%%

red/ax/left: proc D H ⊸
             proc (ax H) C ⊸
             { proc D C }.

red/ax/right: proc (ax H) C ⊸
              proc (Q C) C' ⊸
              { proc (Q H) C' }.

red/cut : proc (cut P (λx. Q x)) C ⊸
          { Exists a.  proc P a ⊗
            proc (Q a) C
          }.

% Multiplicative fragment
red/one : proc oneR C ⊸
          proc (oneL D C) C' ⊸
          { proc D C' }.

red/comm : proc (tensR P1 P2) H ⊸
           proc (tensL (λu1. λu2. Q u1 u2) H) C'' ⊸
           { Exists a. proc P1 a ⊗
             Exists b. proc P2 b ⊗
             proc (Q a b) C''
           }.

red/lolli : proc (lolliR (λu. P u)) C ⊸
            proc (lolliL Q1 (λv. Q2 v) C) C'' ⊸
            { Exists a. proc Q1 a ⊗
              Exists b. proc (P a) b ⊗
              proc (Q2 b) C''
            }.

% Additive fragment
red/with1 : proc (withR ⟨ P1 , P2 ⟩) C ⊸
            proc (withL1 (λv. Q v) C) C'' ⊸
            { Exists a. proc P1 a ⊗
              proc (Q a) C''
            }.

red/with2 : proc (withR ⟨ P1 , P2 ⟩) C ⊸
            proc (withL2 (λv. Q v) C) C'' ⊸
            { Exists a. proc P2 a ⊗
              proc (Q a) C''
            }.

red/plus1 : proc (plusR1 P) C ⊸
             proc (plusL ⟨ (λv. Q1 v), (λv. Q2 v) ⟩ C) C'' ⊸
            { Exists a. proc P a ⊗
              proc (Q1 a) C''
```

```
            }.

red/plus2 : proc (plusR2 P) C ⊸
             proc (plusL ⟨ (λv. Q1 v), (λv. Q2 v) ⟩ C) C'' ⊸
            { Exists a. proc P a ⊗
              proc (Q2 a) C''
            }.
```

Focused Session Types in Celf

```
% Types and Judgments
o    : type.
conc : o → type.
hyp  : o → type.
focL : o → o → type.
focR : o → type.

proc  : conc A → hyp A → type.
procR : focR A → hyp A → type.
procL : hyp A → focL A C → hyp C → type.

% Formulas
one   : o.
tens  : o → o → o.
lolli : o → o → o.
with  : o → o → o.
plus  : o → o → o.

%%%%%%%%%%
% Typing %
%%%%%%%%%%

axR     : hyp A ⊸ focR A.
axL     : focL A A.
cut     : conc A ⊸ (hyp A ⊸ conc C) ⊸ conc C.
focusL  : focL A B ⊸ (hyp A ⊸ conc B).
focusR  : focR A ⊸ conc A.
blurL   : (hyp A ⊸ conc B) ⊸ focL A B.
blurR   : conc A ⊸ focR A.

% The Multiplicative Fragment
oneR    : focR one.
oneL    : conc C ⊸ (hyp one ⊸ conc C).
tensR   : focR A ⊸ focR B ⊸ focR (tens A B).
tensL   : (hyp A ⊸ hyp B ⊸ conc C) ⊸ (hyp (tens A B) ⊸ conc C).
lolliR  : (hyp A ⊸ conc B) ⊸ (conc (lolli A B)).
lolliL  : focR A ⊸ focL B C ⊸ (focL (lolli A B) C).

% The Additive Fragment
withR  : conc A & conc B ⊸ conc (with A B).
withL1 : focL A C ⊸ focL (with A B) C.
withL2 : focL B C ⊸ focL  (with A B) C.
plusR1 : focR A ⊸ focR (plus A B).
plusR2 : focR B ⊸ focR (plus A B).
plusL  : (hyp A ⊸ conc C) & (hyp B ⊸ conc C) ⊸ (hyp (plus A B) ⊸ conc C).

%%%%%%%%%%%%%%
% Reductions %
%%%%%%%%%%%%%%

red/ax/right : proc P C ⊸
```

```
               procR (axR C) C'' ⊸
               { proc P C'' }.

red/ax/rightL : procL H P C ⊸
               procR (axR C) C'' ⊸
               { procL H P C'' }.

red/ax/rightR : procR P C ⊸
              procR (axR C) C'' ⊸
              { procR P C'' }.

red/ax/left : proc P H ⊸
              procL H axL C ⊸
              { proc P C }.

red/ax/leftL : procL H' P H ⊸
              procL H axL C ⊸
              { procL H' P C }.

red/ax/leftR : procR P H ⊸
              procL H axL C ⊸
              { procR P C }.

red/cut : proc (cut P (λx. Q x)) C ⊸
          { Exists a.  proc P a ⊗
            proc (Q a) C
          }.

red/focusR : proc (focusR P) C ⊸
             { procR P C }.

red/focusL : proc (focusL P C) C'' ⊸
             { procL C P C'' }.

red/blurR : procR (blurR P) C ⊸
            { proc P C }.

red/blurL : procL C (blurL (λu. P u)) C'' ⊸
            { proc (P C) C'' }.

% Multiplicative fragment
red/comm : procR (tensR P1 P2) C ⊸
           proc (tensL (λu1. λu2. Q u1 u2) C) C'' ⊸
           { Exists a. procR P1 a ⊗
             Exists b. procR P2 b ⊗
             proc (Q a b) C''
           }.

red/one : procR oneR C ⊸
          proc (oneL D C) C'' ⊸
          { proc D C''
          }.

red/lolli : proc (lolliR (λu. P u)) C ⊸
            procL C (lolliL Q1 Q2)  C'' ⊸
            { Exists a. procR Q1 a ⊗
              Exists b. proc (P a) b ⊗
              procL b Q2 C''
            }.

% Additive fragment
```

```
red/with1 : proc (withR ⟨ P1 , P2 ⟩) C ⊸
            procL C (withL1 Q) C'' ⊸
            { Exists a. proc P1 a ⊗
              procL a Q C''
            }.

red/with2 : proc (withR ⟨ P1 , P2 ⟩) C ⊸
            procL C (withL2 Q) C'' ⊸
            { Exists a. proc P2 a ⊗
              procL a Q C''
            }.

red/plus1 : procR (plusR1 P) C ⊸
              proc (plusL ⟨ (λv. Q1 v) ,
                            (λv. Q2 v) ⟩ C) C'' ⊸
             { Exists a. procR P a ⊗
               proc (Q1 a) C''
             }.

red/plus2 : procR (plusR2 P) C ⊸
              proc (plusL ⟨ (λv. Q1 v) ,
                            (λv. Q2 v) ⟩ C) C'' ⊸
             { Exists a. procR P a ⊗
               proc (Q2 a) C''
             }.
```