LFM 2004 Preliminary Version

Specifying Properties of Concurrent Computations in CLF [⋆]

Kevin Watkins [a,1] Iliano Cervesato [b,2] Frank Pfenning [a,3]
David Walker [c,4]

[a] *Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA*

[b] *ITT Industries, AES Division, Alexandria, VA*

[c] *Department of Computer Science, Princeton University, Princeton, NJ*

Abstract

CLF (the Concurrent Logical Framework) is a language for specifying and reasoning about concurrent systems. Its most significant feature is the first-class representation of *concurrent executions as monadic expressions*. We illustrate the representation techniques available within CLF by applying them to an asynchronous pi-calculus with correspondence assertions, including its dynamic semantics, safety criterion, and a type system with latent effects due to Gordon and Jeffrey.

Key words: logical frameworks, type theory, linear logic, concurrency

1 Introduction

This paper cannot describe the CLF framework in detail; a complete description is available in other work [19,18,3], and the syntax and typing rules of the framework are summarized in Appendix B. Nevertheless, in this introduction, we briefly discuss the lineage of frameworks on which CLF is based, and the basic design of CLF.

A logical framework is a meta-language for the specification and implementation of deductive systems, which are used pervasively in logic and the theory of programming languages. A logical framework should be as simple

⋆ This research was sponsored in part by the NSF under grants CCR-9988281, CCR-0208601, CCR-0238328, and CCR-0306313, and by NRL under grant N00173-00-C-2086.

1 Email: `kw@cs.cmu.edu`

2 Email: `iliano@itd.nrl.navy.mil`

3 Email: `fp@cs.cmu.edu`

4 Email: `dpw@cs.princeton.edu`

and uniform as possible, yet provide intrinsic means for representing common concepts and operations in its application domain. A logical framework is characterized by an underlying meta-logic or type theory and a representation methodology.

The principal starting point for our work is the LF logical framework [7], which is based on a minimal type theory λ^{Π} with only the dependent function type constructor Π. LF directly supports concise expression of variable renaming and capture-avoiding substitution at the level of syntax, and parametric and hypothetical judgments in deductions, following the *judgments-as-types* principle. Proofs are reified as objects, which allows properties of and relations between proofs to be expressed within the framework.

Representations of systems involving state remained cumbersome until the design of the linear logical framework LLF [2] and its close relative RLF [10]. LLF is a conservative extension of LF with the linear function type $A \multimap B$, the additive product type $A \mathbin{\&} B$, and the additive unit type $\top$. The main additional representation of LLF is that of *state-as-linear-hypotheses*. Imperative computations consequently become linear objects in the framework. They can serve as index objects, which means we can express properties of stateful systems at a high level of abstraction.

While LLF solves many problems associated with stateful computation, the encoding of *concurrent computations* remained unsatisfactory for several reasons. One of the problems is that LLF formulations of concurrent systems inherently sequentialize the computation steps.

In this paper we are concerned with CLF, a conservative extension of LLF with intrinsic support for concurrency. Concurrent computations are encapsulated in a monad [15], which permits a simple definitional equality and guarantees conservativity over LF and LLF. The definitional equality on monadic expressions identifies different interleavings of independent steps, thereby expressing *true concurrency*. Dependent types then allow us to specify properties of concurrent computations, as long as they do not rely on the order of independent steps.

We illustrate the framework's expressive power and representation techniques through a sample encoding of the asynchronous π-calculus with correspondence assertions, following Gordon and Jeffrey [6]. Further examples, such as encodings of Petri-nets, Concurrent ML, and the security protocol specification framework MSR can be found in another technical report [3].

The remainder of the paper is organized as follows: Section 2 introduces the π-calculus with which we are concerned and its syntax; Section 3 describes the original static semantics of Gordon et al. and its CLF representation; Section 4 describes the operational semantics of the language and its CLF representation; Section 5 introduces the syntax of traces and describes the abstraction judgment relating computations and traces, and briefly discusses the safety criterion; Section 6 briefly describes related work; and Section 7 concludes. Appendices summarize the π-calculus encoding and the syntax and judgments

$$
\begin{array}{rcl}
\mathsf{pr} & : & \mathsf{type.}\\
\mathsf{nm} & : & \mathsf{type.}\\
\mathsf{tp} & : & \mathsf{type.}\\
\mathsf{label} & : & \mathsf{type.}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{stop} & : & \mathsf{pr.} & \ulcorner\mathsf{stop}\urcorner & = & \mathsf{stop}\\
\mathsf{par} & : & \mathsf{pr} \rightarrow \mathsf{pr} \rightarrow \mathsf{pr.} & \ulcorner P \mid Q\urcorner & = & \mathsf{par}\,\ulcorner P\urcorner\,\ulcorner Q\urcorner\\
\mathsf{repeat} & : & \mathsf{pr} \rightarrow \mathsf{pr.} & \ulcorner\mathsf{repeat}\ P\urcorner & = & \mathsf{repeat}\,\ulcorner P\urcorner\\
\mathsf{new} & : & \mathsf{tp} \rightarrow (\mathsf{nm} \rightarrow \mathsf{pr}) \rightarrow \mathsf{pr.} & \ulcorner\mathsf{new}(x{:}\tau); P\urcorner & = & \mathsf{new}\,\ulcorner\tau\urcorner\,(\lambda x.\ulcorner P\urcorner)\\
\mathsf{choose} & : & \mathsf{pr} \rightarrow \mathsf{pr} \rightarrow \mathsf{pr.} & \ulcorner\mathsf{choose}\ P\ Q\urcorner & = & \mathsf{choose}\,\ulcorner P\urcorner\,\ulcorner Q\urcorner\\
\mathsf{out} & : & \mathsf{nm} \rightarrow \mathsf{nm} \rightarrow \mathsf{pr.} & \ulcorner\mathsf{out}\ x\langle y\rangle\urcorner & = & \mathsf{out}\ x\ y\\
\mathsf{inp} & : & \mathsf{nm} \rightarrow \mathsf{tp} \rightarrow & \ulcorner\mathsf{inp}\ x(y{:}\tau); P\urcorner & = & \mathsf{inp}\ x\,\ulcorner\tau\urcorner\,(\lambda y.\ulcorner P\urcorner)\\
 & & \quad (\mathsf{nm} \rightarrow \mathsf{pr}) \rightarrow \mathsf{pr.} & & & \\
\mathsf{begin} & : & \mathsf{label} \rightarrow \mathsf{pr} \rightarrow \mathsf{pr.} & \ulcorner\mathsf{begin}\ L; P\urcorner & = & \mathsf{begin}\,\ulcorner L\urcorner\,\ulcorner P\urcorner\\
\mathsf{end} & : & \mathsf{label} \rightarrow \mathsf{pr} \rightarrow \mathsf{pr.} & \ulcorner\mathsf{end}\ L; P\urcorner & = & \mathsf{end}\,\ulcorner L\urcorner\,\ulcorner P\urcorner
\end{array}
$$

Fig. 1. Process syntax represented in CLF

of the framework.

2 The asynchronous π-calculus with correspondence assertions

Our asynchronous π-calculus with correspondence assertions follows Gordon and Jeffrey's presentation [6]. Correspondence assertions, originally developed by Woo and Lam [20], come in two varieties, begin L and end L, where L is a label that carries information about the state of a communication protocol. Gordon and Jeffrey have shown that a variety of important correctness properties of cryptographic protocols can be stated in terms of matching pairs of these begin and end assertions.

To illustrate the basic ideas, we will examine a simple handshake protocol taken directly from Gordon and Jeffrey's work. This protocol is intended to ensure that if a sender named a receives an acknowledgment message then the receiver named b has actually received the message. In the asynchronous π-calculus with correspondence assertions, we specify the protocol as follows.

$$
\begin{array}{rcl}
\mathrm{Send}(a,b,c) & = & \mathsf{new}(msg); \mathsf{new}(ack);\\
 & & \quad (\mathsf{out}\ c\langle msg, ack\rangle\\
 & & \quad\ \mid \mathsf{inp}\ ack(); \mathsf{end}\ (a,b,msg))\\
\mathrm{Rcv}(a,b,c) & = & \mathsf{inp}\ c(x,y); \mathsf{begin}\ (a,b,x); \mathsf{out}\ y\langle\rangle
\end{array}
$$

The standard π-calculus process constructors used here are parallel composition ($P \mid Q$), generation of a new name x to be used in a process P ($\mathsf{new}(x); P$ where x is bound in P), asynchronous output on channel c ($\mathsf{out}\ c\langle msg, ack\rangle$), and input on channel c ($\mathsf{inp}\ c(x_1, \ldots, x_n); P$ where variables x_1 through x_n are bound in P). In the protocol, the sending process generates a new message and a new acknowledgment channel. The sender uses the asynchronous output command to send the pair of message and acknowledgment channel on c and waits for a response on ack. Once the sender receives the acknowledgment, it executes an end assertion which specifies that the sender (named a) requires that the receiver (named b) has already received the input message (msg). The receiver cooperates with the sender by waiting for pairs of message and acknowledgment on channel c. After receiving on c, the begin assertion declares that the receiver b has received the input message. After this declaration, the receiver sends an acknowledgment to the sender. We hope that in all executions of senders in parallel with receivers, begin assertions match up with end assertions. If they do, sender a can be sure that receiver b received the message msg.

Now, consider combining a single sender in parallel with a single receiver: $\mathsf{new}(c); (\mathrm{Send}(a, b, c) \mid \mathrm{Rcv}(a, b, c))$. This configuration is *safe* since in every possible execution, every end (a, b, msg) assertion is preceded in that execution by a distinct corresponding begin (a, b, msg) assertion. On the other hand, placing multiple different senders in parallel with a single copy of a receiver is *unsafe*:

$$\mathrm{Send}(a, b, c) \mid \mathrm{Send}(a', b, c) \mid \mathrm{Rcv}(a, b, c)$$

This configuration is *unsafe* because there exists an execution in which an end L assertion is executed but there has been no prior matching begin L. More specifically, the second sender a' may create a message and send it to the receiver. The receiver, thinking it is communicating with a, receives the message, executes begin (a, b, msg), and returns the acknowledgment. Finally, the second sender executes end (a', b, msg). In this protocol, since the identity of the sender (either a or a') was not included in the message, there has been confusion over who the receiver was communicating with. This is a very simple example, but Gordon and Jeffrey have demonstrated that these assertions can be used to identify serious flaws in much more complicated and important protocols.

2.1 Syntax

The syntax of the π-calculus processes P with correspondence assertions is presented below. We have simplified Gordon and Jeffrey's calculus in a couple of ways, replacing polyadic input and output processes with monadic versions, dropping any data structures other than channels x, y, z and replacing deterministic if statements with non-deterministic choice (choose P Q). Two process forms that did not show up in the informal example above are the process stop, which does nothing, and the replicated process repeat P, which acts as an unbounded number of copies of itself. The static semantics makes use of types τ, which are discussed in the next section; these do not affect the operational semantics of a program.

$$\begin{array}{rcl} \mathsf{name} & : & \mathsf{tp}. \\ \mathsf{chan} & : & \mathsf{tp} \to (\mathsf{nm} \to \mathsf{eff}) \to \mathsf{tp}. \end{array} \qquad \begin{array}{rcl} \ulcorner \mathsf{Name} \urcorner & = & \mathsf{name} \\ \ulcorner \mathsf{Ch}(x\!:\!\tau)e \urcorner & = & \mathsf{chan}\, \ulcorner \tau \urcorner\, (\lambda x. \ulcorner e \urcorner) \end{array}$$

Fig. 2. Type syntax represented in CLF

$$\begin{array}{rl} P, Q ::= & \mathsf{stop} \mid (P \mid Q) \mid \mathsf{repeat}\ P \mid \mathsf{new}(x\!:\!\tau); P \\ & \mid \mathsf{choose}\ P\ Q \mid \mathsf{out}\ x\langle y \rangle \mid \mathsf{inp}\ x(y\!:\!\tau); P \\ & \mid \mathsf{begin}\ L; P \mid \mathsf{end}\ L; P \end{array}$$

The representation of process syntax follows standard LF methodology. The signature, shown in Figure 1, represents process syntax via CLF types pr (processes), nm (names), tp (types), and label (assertion labels). The representation function mapping processes to CLF objects is shown at the right.

A few comments: The type nm of names does not contain any closed terms; it classifies bound variables within a process expression. The type tp is discussed in Section 3. Channels are a special case of names. We do not specify any particular syntax for assertion labels, but it is assumed that they might mention names bound by new or inp. As is common in LF representations, we use *higher-order abstract syntax*, which allows us to model π-calculus bound variables using framework variables and to implement π-calculus substitution using the framework's substitution.

The most important property of this representation is *adequacy*: every process in the original language has its own representative as a CLF object of type pr, and every object in pr is such a representation. The canonical forms property for CLF renders proofs of such results almost trivial.

3 The static semantics

Gordon and Jeffrey present a static semantics with types and effects for their language. The goal of the static semantics is to ensure that the *correspondence property* for assertions is not violated: for each end L assertion reached in an execution, a distinct begin L assertion for L must have been reached in the past. The static semantics associates an effect e (a multiset of labels) with each program point, such that it is safe to execute end L for each label L in the multiset. (Of course, not all safe programs will necessarily have a valid typing.)

Since CLF includes LLF as a sublanguage, we will be able to represent the static "state" of the effect system as a multiset of linear hypotheses in LLF style [2]. The basic idea is to record a multiset of begins already reached at the current program point as linear hypotheses of the typing judgment.[5] Then each occurrence of begin L contributes a linear hypothesis of type effect L for

[5] Really these are *affine* hypotheses, since the invariant is that the multiset be merely a lower bound: it is perfectly safe to "forget" that a begin was reached at some point in the past. Careful use of the additives $\top$ and & will allow us to simulate affine hypotheses with linear ones.

$$
\begin{array}{rcl}
\mathsf{has} & : & \mathsf{nm} \rightarrow \mathsf{tp} \rightarrow \mathsf{type}. \\
\mathsf{good} & : & \mathsf{pr} \rightarrow \mathsf{type}. \\
\mathsf{consume} & : & \mathsf{eff} \rightarrow \mathsf{type}. \\
\mathsf{assume} & : & \mathsf{eff} \rightarrow \mathsf{pr} \rightarrow \mathsf{type}. \\
\\
\mathsf{gd\_stop} & : & \mathsf{good}\ \mathsf{stop} \multimapinv \top. \\
\mathsf{gd\_par} & : & \mathsf{good}\ (\mathsf{par}\ P\ Q) \multimapinv \mathsf{good}\ P \multimapinv \mathsf{good}\ Q. \\
\mathsf{gd\_repeat} & : & \mathsf{good}\ (\mathsf{repeat}\ P) \multimapinv \top \leftarrow \mathsf{good}\ P. \\
\mathsf{gd\_new} & : & \mathsf{good}\ (\mathsf{new}\ \tau\ (\lambda x.\, P\ x)) \leftarrow \mathsf{wftp}\ \tau \\
& & \quad \multimapinv (\Pi x\!:\!\mathsf{nm}.\, \mathsf{has}\ x\ \tau \rightarrow \mathsf{good}\ (P\ x)). \\
\mathsf{gd\_choose} & : & \mathsf{good}\ (\mathsf{choose}\ P\ Q) \multimapinv (\mathsf{good}\ P\ \&\ \mathsf{good}\ Q). \\
\mathsf{gd\_out} & : & \mathsf{good}\ (\mathsf{out}\ X\ Y) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \leftarrow \mathsf{has}\ Y\ \tau \\
& & \quad \multimapinv \mathsf{consume}\ (E\ Y). \\
\mathsf{gd\_inp} & : & \mathsf{good}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \leftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \\
& & \quad \leftarrow (\Pi y\!:\!\mathsf{nm}.\, \mathsf{has}\ y\ \tau \rightarrow \mathsf{assume}\ (E\ y)\ (P\ y)). \\
\mathsf{gd\_begin} & : & \mathsf{good}\ (\mathsf{begin}\ L\ P) \multimapinv (\mathsf{effect}\ L \multimap \mathsf{good}\ P). \\
\mathsf{gd\_end} & : & \mathsf{good}\ (\mathsf{end}\ L\ P) \multimapinv \mathsf{effect}\ L \multimapinv \mathsf{good}\ P. \\
\\
\mathsf{con\_eps} & : & \mathsf{consume}\ \{1\} \multimapinv \top. \\
\mathsf{con\_join} & : & \mathsf{consume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\} \\
& & \quad \multimapinv \mathsf{effect}\ L \multimapinv \mathsf{consume}\ E. \\
\\
\mathsf{ass\_eps} & : & \mathsf{assume}\ \{1\}\ P \multimapinv \mathsf{good}\ P. \\
\mathsf{ass\_join} & : & \mathsf{assume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\} \\
& & \quad \multimapinv (\mathsf{effect}\ L \multimap \mathsf{assume}\ E\ P).
\end{array}
$$

Fig. 3. Static semantics represented in CLF

the checking of its continuation, and each end L consumes such a hypothesis.

This accounts for trivial instances of correct programs in which an end is found directly within the continuation of its matching begin. Of course, in actual use, one is more interested in cases in which the end and its matching begin occur in different processes executing concurrently (as in the example of Section 2).

Gordon et al. introduce *latent effects* to treat many such cases. The idea is that each value transmitted across a channel may carry with it a multiset of latent effects, the effects being *debited* from the process sending the value and *credited* to the process receiving it. Since communication synchronizes the sending and receiving processes, it is certain that the begins introducing the debited effects in the sending process will occur before any ends making use of the credited effects in the receiving process.[6]

These considerations lead to a simple type syntax. Each name in the static semantics has a type τ: either Name (really nonsense; i.e., just a nonce) or

[6] Of course, this implicitly relies on the unicast nature of communication in the language. If multicast or broadcast were allowed, more than one process could be credited, violating the non-duplicable nature of effect hypotheses.

$\mathsf{Ch}(x\!:\!\tau)e$, representing a channel transmitting names of type τ and a latent effect e. These types ("π-types," for short) are represented by CLF type tp, the constructors of which are shown in Figure 2. Latent effects e are themselves multisets of labels, and are represented in CLF by a type eff discussed below.

Although a latent effect is again a multiset of labels, the LLF strategy of representing multisets by linear hypotheses does not apply, because latent effects must be first-class values. An LF strategy using explicit list constructors (cons and nil) would represent the latent effects as first-class values, but the LF equality on such lists would be too restrictive: $[L_1, L_2]$ and $[L_2, L_1]$ are equal as multisets, but $\mathsf{cons}\ L_1\ (\mathsf{cons}\ L_2\ \mathsf{nil})$ and $\mathsf{cons}\ L_2\ (\mathsf{cons}\ L_1\ \mathsf{nil})$ are not necessarily equal as LF objects.

In CLF, we have a new alternative: expressions are first-class objects, and CLF's concurrent equality on them can model multiset equality precisely. Each label multiset $[L_1, \ldots, L_n]$ will be represented by an expression $\{\mathsf{let}\ \{1\} = \mathsf{latent}\ L_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ \{1\} = \mathsf{latent}\ L_n\ \mathsf{in}\ 1\}$. The equality on the representation then naturally models equality of multisets. We take eff to be a notational abbreviation for the type $\{1\}$ of such expressions, and add the following declaration to the signature.

$$\mathsf{latent}\ :\ \mathsf{label} \to \{1\}.$$

In addition, we must axiomatize the objects of type $\{1\}$ that correspond to such multisets; this is the judgment wfeff presented in Appendix A.

Next we represent the π-calculus typing judgment itself as a CLF type family good, defined in Figure 3. We use $A \multimapinv B$ and $A \leftarrow B$, which associate to the left, as alternate forms of $B \multimap A$ and $B \to A$, giving the signature the shape of a logic program. The type A in $\Pi u\!:\!A.\,B$ has been omitted where it is determined by context. We often omit outermost Π quantifiers; in such cases the corresponding arguments to the constant in question are also omitted (implicit). We have also η-contracted some subterms in order to conserve space; these should be read as abbreviations for their η-long (canonical) forms.

Since not every declared effect must actually occur (that is, there is implicitly a *weakening* principle for effects), we must use the additive unit $\top$ to consume any leftover effects at the leaves of a derivation (instances of the $\mathsf{gd\_stop}$ or $\mathsf{con\_eps}$ rules).

The type family wftp, not shown in the figure (see Appendix A), represents the judgment that a π-type is well formed, reducing more or less to the judgment wfeff for any latent effects mentioned in the π-type. The type family has contains no closed objects, but in the course of a derivation of $\mathsf{good}\ P$, hypotheses $\mathsf{has}\ x\ \tau$ will be introduced for each name bound by new or inp in P. Similarly, the family effect has no closed objects, but in the course of a typing derivation, linear hypotheses $\mathsf{effect}\ L$ can be introduced by begin and consumed by end.

The task of assume and $\mathsf{consume}$ is to introduce and consume linear hypotheses for the whole multiset of effects contained in a latent effect. Latent effects are consumed by out, which has no continuation, and produced by inp, which does. Accordingly, assume takes the continuation as an argument, and invokes good to check it once the multiset of effects has been introduced into the linear context.

It can be shown by extensions of the standard techniques developed for the LLF fragment of CLF that this representation is *adequate*: a process P is well-typed in the original system just when there is an object of type good P in CLF.

4 The operational semantics

Gordon and Jeffrey's operational semantics [6] is based on a traced transition system $P \xrightarrow{s} P'$, where s is a trace: a sequence of begin and end actions, internal actions τ, and gen actions binding freshly generated names (corresponding to the execution of new). Although we have not specified the language of labels, it is assumed that they may mention such names. Then $P \xrightarrow{s} P'$ when P can evolve to P' while performing the actions in trace s. The traced transition system itself depends on the usual notion of structural congruence $P \equiv P'$ found in the π-calculus literature.

The CLF representation has a somewhat different structure. Since CLF has a first-class notation for concurrent computations, we can factor the traced transition system into two judgments: first, that a process P has a concurrent execution E (which is represented by a CLF expression); and second, that an execution E has a (serialized) trace s. This section is concerned with the first judgment, while the next section treats traces.

Computations in this semantics are represented by CLF expressions

$$x_1{:}\mathsf{nm}, \ldots, x_m{:}\mathsf{nm},\ r_1{:}\mathsf{run}\ P_1, \ldots, r_i{:}\mathsf{run}\ P_i; \\ r_{i+1}\hat{:}\mathsf{run}\ P_{i+1}, \ldots, r_n\hat{:}\mathsf{run}\ P_n\ \vdash\ E \leftarrow \top$$

in a context having unrestricted hypotheses of type nm for each generated name, unrestricted hypotheses $r_1 \ldots r_i$ of type run P for each process P that is executing and available unrestrictedly, and linear hypotheses $r_{i+1} \ldots r_n$ of type run P for each process P that is available linearly, where run : pr $\rightarrow$ type.[7] The final result of the computation is taken as the additive unit $\top$, which means that computation can stop at any time, with any leftover resources (linear hypotheses) consumed by $\langle\rangle$, its introduction form.

Then each of the *structural* process constructors stop, par, repeat, and new can be represented by a corresponding synchronous CLF connective:

[7] Here $\leftarrow$ denotes the lax typing judgment, not reverse implication.

$$
\begin{aligned}
\mathsf{ev\_stop} &: \mathsf{run\ stop} \multimap \{1\}. \\
\mathsf{ev\_par} &: \mathsf{run}\ (\mathsf{par}\ P\ Q) \multimap \{\mathsf{run}\ P \otimes \mathsf{run}\ Q\}. \\
\mathsf{ev\_repeat} &: \mathsf{run}\ (\mathsf{repeat}\ P) \multimap \{!\mathsf{run}\ P\}. \\
\mathsf{ev\_new} &: \mathsf{run}\ (\mathsf{new}\ \tau\ (\lambda u.\, P\ u)) \multimap \{\exists u{:}\mathsf{nm}.\, \mathsf{run}\ (P\ u)\}.
\end{aligned}
$$

The remaining constructors are interpreted according to their semantics:

$$
\begin{aligned}
\mathsf{ev\_choose}_i &: \mathsf{run}\ (\mathsf{choose}\ P_1\ P_2) \multimap \{\mathsf{run}\ P_i\}. \\
\mathsf{ev\_sync} &: \mathsf{run}\ (\mathsf{out}\ X\ Y) \multimap \mathsf{run}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \\
&\quad \multimap \{\mathsf{run}\ (P\ Y)\}. \\
\mathsf{ev\_begin} &: \Pi L{:}\mathsf{label}.\, \mathsf{run}\ (\mathsf{begin}\ L\ P) \multimap \{\mathsf{run}\ P\}. \\
\mathsf{ev\_end} &: \Pi L{:}\mathsf{label}.\, \mathsf{run}\ (\mathsf{end}\ L\ P) \multimap \{\mathsf{run}\ P\}.
\end{aligned}
$$

We depart from the usual practice of leaving outermost Π quantifiers implicit for reasons that will become clear in Section 5.

One interesting feature of the CLF encoding is that many of the structural equivalences of the original presentation of the π-calculus appear automatically (shallowly) as consequences of the principles of exchange, weakening (since $\top$ is present) and so forth satisfied by CLF hypotheses. In the CLF setting the rest of the structural equivalences are captured within a general notion of simulation, discussed briefly in Section 5.

In this representation, each concurrent computation starting from a process P corresponds to a CLF object of type $\mathsf{run}\ P \multimap \{\top\}$; that is, a term $\hat{\lambda} r.\, \{E\}$ where E is a monadic expression of type $\top$ in a context containing a single linear hypothesis r representing the process P. Because CLF's notion of equality identifies monadic expressions differing only in the order of execution of independent computation steps, each such object (modulo equality) represents the dependence graph of a possible computation. Thus judgments (represented by CLF types) concerning such objects, such as the abstraction judgment to be introduced in the next section, are predicates on depedence graphs, not on serialized computations.

There is no simple adequacy result at this stage, since the judgment $P \xrightarrow{s} P'$ of Gordon et al. refers to the trace s, which is not directly available in the CLF operational semantics. (Moreover, the process P' to which P evolves is only available in CLF implicitly as the set of leftover hypotheses consumed by the $\top$ introduction at the end of the CLF expression representing a computation.) Once traces and the abstraction judgment relating a computation to its traces are introduced, it will be possible to state a precise adequacy result.

5 Traces and abstraction

The syntax of the traces s mentioned in the judgment $P \xrightarrow{s} P'$ of Gordon et al. can be represented straightforwardly by LF techniques. Though we have left the label syntax unspecified, it is assumed that labels might depend on names

$$
\begin{array}{rcl}
\mathsf{abst} & : & \{\top\} \rightarrow \mathsf{tr} \rightarrow \mathsf{type}. \\[1ex]
\mathsf{abst\_nil} & : & \mathsf{abst}\ E\ \mathsf{tnil}. \\
\mathsf{abst\_stop} & : & \mathsf{abst}\ \{\mathsf{let}\ \{1\} = \mathsf{ev\_stop}\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ \mathsf{in}\ \langle\rangle\}\ s \leftarrow \mathsf{abst}\ E\ s. \\
\mathsf{abst\_par} & : & \mathsf{abst}\ \{\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev\_alt}\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\hat{}r_1\hat{}r_2\ \mathsf{in}\ \langle\rangle\}\ s \\
 & & \quad \leftarrow (\Pi r_1.\, \Pi r_2.\, \mathsf{abst}\ (E\hat{}r_1\hat{}r_2)\ s). \\
\mathsf{abst\_repeat} & : & \mathsf{abst}\ \{\mathsf{let}\ \{!r\} = \mathsf{ev\_repeat}\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ r\ \mathsf{in}\ \langle\rangle\}\ s \\
 & & \quad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s). \\
\mathsf{abst\_new} & : & \mathsf{abst}\ \{\mathsf{let}\ \{[x, r]\} = \mathsf{ev\_new}\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ x\hat{}r\ \mathsf{in}\ \langle\rangle\} \\
 & & \qquad (\mathsf{tgen}\ (\lambda x.\, s\ x)) \\
 & & \quad \leftarrow (\Pi x.\, \Pi r.\, \mathsf{abst}\ (E\ x\hat{}r)\ (s\ x)). \\
\mathsf{abst\_choose}_i & : & \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_choose}_i\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s) \\
 & & \quad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s). \\
\mathsf{abst\_sync} & : & \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_sync}\hat{}R_1\hat{}R_2\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s) \\
 & & \quad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s). \\
\mathsf{abst\_begin} & : & \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_begin}\ L\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tbegin}\ L\ s) \\
 & & \quad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s). \\
\mathsf{abst\_end} & : & \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_end}\ L\hat{}R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\hat{}r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tend}\ L\ s) \\
 & & \quad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\hat{}r)\ s).
\end{array}
$$

Fig. 4. The abstraction judgment as a CLF program

introduced in the course of the computation, so the actions gen representing the generation of fresh names in the execution of a new process must bind a variable in the style of higher-order abstract syntax.

The representation of traces is as follows:

$$
\begin{array}{rcll}
\mathsf{tr} & : & \mathsf{type}. & \\[1ex]
\mathsf{tnil} & : & \mathsf{tr}. & \ulcorner \varepsilon \urcorner = \mathsf{tnil} \\
\mathsf{tint} & : & \mathsf{tr} \rightarrow \mathsf{tr}. & \ulcorner \tau, s \urcorner = \mathsf{tint}\ \ulcorner s \urcorner \\
\mathsf{tbegin} & : & \mathsf{label} \rightarrow \mathsf{tr} \rightarrow \mathsf{tr}. & \ulcorner \mathsf{begin}\ L, s \urcorner = \mathsf{tbegin}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\mathsf{tend} & : & \mathsf{label} \rightarrow \mathsf{tr} \rightarrow \mathsf{tr}. & \ulcorner \mathsf{end}\ L, s \urcorner = \mathsf{tend}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\mathsf{tgen} & : & (\mathsf{nm} \rightarrow \mathsf{tr}) \rightarrow \mathsf{tr}. & \ulcorner \mathsf{gen}\ \langle x \rangle, s \urcorner = \mathsf{tgen}\ (\lambda x.\, \ulcorner s \urcorner)
\end{array}
$$

Now we are equipped with enough tools to write the abstraction judgment relating a computation to its traces, as a CLF type family $\mathsf{abst} : \{\top\} \rightarrow \mathsf{tr} \rightarrow \mathsf{type}$, the logic program for which is shown in Figure 4. The first argument of this relation is the CLF object representing the dependence graph of the computation, while the second argument is an associated trace. The mode (in the sense of logic programming) is input for the first argument and output for the second. However, the relation is not a function, because from a single execution (as dependence graph) many possible (serial) abstractions as a trace might be extracted. Nevertheless, each execution has at least one abstraction as a trace.

It is also noteworthy that the context in which the abst judgment executes uses unrestricted hypotheses $r : \mathsf{run}\ P$ for each executing process P, whether or not the corresponding process was represented by a linear hypothesis in the original execution. This is a common phenomenon when writing higher-level judgments in LLF style.

This judgment, taken together with CLF's equality admitting concurrency equations, defines for each concurrent computation the set of possible serializations of that computation as a trace. The traces need not describe the whole computation; the rule abst_nil allows abstraction to stop after computing the trace of any prefix of the computation. This suffices because we are interested only in *safety* properties, which are violated whenever they are violated on a prefix of the computation.

We would like to show that each traced transition $P \xrightarrow{s} P'$ of Gordon and Jeffrey's system corresponds to an object $\hat{\lambda} r.\, E : \mathsf{run}\ P \multimap \{\top\}$ as in Section 4 together with an abstraction $\mathsf{abst}\ E\ s$ yielding the same trace. As it turns out, this is not quite the case, because the structural equivalences considered in that paper induce certain rearrangements of tgen with respect to other actions that are not possible in the CLF variant. However, defining an appropriate notion of "similarity" on traces admitting rearrangement of tgen steps (which, moreover, can be characterized by another CLF judgment), we find that each traced transition is in correspondence with a CLF expression and abstraction yielding a "similar" trace.

The proof technique is illustrative but is not presented here in detail. In brief, one considers the notion of simulation $P_1 \preceq P_2$ induced by the CLF operational semantics of Section 4, abstraction, and "similarity" of traces: whenever P_1 and some context consisting of other processes and names yields a given trace, P_2 yields a similar trace in the same context. Then all the structural equivalences of the traced transition system are simulations in this sense, and it follows easily that each traced transition has its CLF counterpart. The converse is simple, because each rule of the CLF operational semantics is immediately available as a step of the traced transition system (or a structural equivalence). So we have:

Proposition 1 (Adequacy of operational semantics) *The traced transition system proves $P \xrightarrow{s} P'$ for some P' just when there exist $E : \mathsf{run}\ \ulcorner P \urcorner \multimap \{\top\}$ and $A : (\Pi r.\, \mathsf{abst}\ (E^\wedge r)\ s')$ (in a context binding the free names of P and P'), and s is similar to s'.*

Finally, we can define the *safety criterion* for processes. In a constructive setting, it is easiest to characterize *unsafety*, because it is witnessed by finitary evidence. A process is unsafe precisely when it has an execution admitting some abstraction as a trace that violates the correspondence property (see Section 2). It turns out to be easy to write a CLF judgment characterizing those traces that violate the correspondence property (see Appendix A). Thus, each step of the criterion is modeled by a CLF judgment, and we can write an overall judgment unsafe P, which, as a CLF type, contains all the proofs of unsafety of P. This turns out to be the same, mutatis mutandis, as Gordon and Jeffrey's definition.

6 Related work

Right from its inception, linear logic [5] has been advocated as a logic with an intrinsic notion of state and concurrency. In the literature, many connections between concurrent calculi and linear logic have been observed. Due to space constraints we cannot survey this relatively large literature here. In a logical framework, we remove ourselves by one degree from the actual semantics; we represent rather than embed calculi. Thereby, CLF provides another point of view on many of the prior investigations.

Most closely related to our work is Miller's logical framework Forum [13], which is based on a sequent calculus for classical linear logic and focusing proofs [1]. As shown by Miller and elaborated by Chirimar [4], Forum can also represent concurrency. Our work extends Forum in several directions. Most importantly, it is a type theory based on natural deduction and therefore offers an internal notion of proof object that is not available in Forum. Among other things, this means we can explicitly represent relations on deductions and therefore on concurrent computations.

There have been several formalizations of versions of the π-calculus in a variety of reasoning systems, such as HOL [11], Coq [8,9], Isabelle/HOL [17] or Linc [14]. A distinguishing feature of our sample encoding in this paper is the simultaneous use of higher-order abstract syntax, linearity, modality, and the intrinsic notion of concurrent computations. Also, we are not aware of a formal treatment of correspondence assertions or dependent effect typing for the π-calculus.

Systems based on rewriting logic, such as Maude [12], natively support concurrent specifications (and have been used to model Petri nets, CCS, the π-calculus, etc). However, lacking operators comparable to CLF's dependent types and proof-terms, Maude users must code concurrent computations independently from the concurrent systems that originate them.

As already mentioned above, CLF is a conservative extension of LLF with the asynchronous connectives $\otimes$, 1, !, and $\exists$, encapsulated in a monad. The idea of monadic encapsulation goes back to Moggi's monadic meta-language [15] and is used heavily in functional programming. Our formulation follows the judgmental presentation of Pfenning and Davies [16] that completely avoids the need for commuting conversions, but treats neither linearity nor the existence of normal forms. This permits us to reintroduce some equations to model true concurrency in a completely orthogonal fashion.

7 Conclusions

The goal of this work has been to extend the elegant and logically motivated representation strategies for syntax, judgments, and state available in LF and

LLF to the concurrent world. We have shown how the availability of a *notation* for concurrent executions, admitting a proper truly concurrent equality, enables powerful strategies for specifying properties of such executions.

Ultimately, it should become as simple and natural to manipulate the objects representing concurrent executions as it is to manipulate LF objects. If higher-order abstract syntax means never having to code up α-conversion or capture-avoiding substitution ever again, we hope that in the same way, the techniques explored here can make it unnecessary to code up multiset equality or concurrent equality ever again, so that intellectual effort can be focused on reasoning about deeper properties of concurrent systems.

References

[1] Andreoli, J.-M., *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), pp. 197–347.

[2] Cervesato, I. and F. Pfenning, *A linear logical framework*, Information & Computation **179** (2002), pp. 19–75.

[3] Cervesato, I., F. Pfenning, D. Walker and K. Watkins, *A concurrent logical framework II: Examples and applications*, Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University (2002), revised May 2003.

[4] Chirimar, J. L., "Proof Theoretic Approach to Specification Languages," Ph.D. thesis, University of Pennsylvania (1995).

[5] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[6] Gordon, A. D. and A. Jeffrey, *Typing correspondence assertions for communication protocols*, Theoretical Computer Science **300** (2003), pp. 379–409.

[7] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery **40** (1993), pp. 143–184.

[8] Hirschkoff, D., *A full formalisation of pi-calculus theory in the Calculus of Constructions*, in: E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'97)* (1997), pp. 153–169.

[9] Honsell, F., M. Miculan and I. Scagnetto, *Pi-calculus in (co)inductive type theories*, Theoretical Computer Science **253** (2001), pp. 239–285.

[10] Ishtiaq, S. and D. Pym, *A relevant analysis of natural deduction*, Journal of Logic and Computation **8** (1998), pp. 809–838.

[11] Melham, T., *A mechanized theory of the pi-calculus in HOL*, Nordic Journal of Computing **1** (1995), pp. 50–76.

[12] Meseguer, J., *Software specification and verification in rewriting logic*, Lecture notes for the Marktoberdorf International Summer School, Germany (2002).

[13] Miller, D., *A multiple-conclusion meta-logic*, in: S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science* (1994), pp. 272–281.

[14] Miller, D. and A. Tiu, *A proof theory for generic judgments*, in: P. Kolaitis, editor, *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)* (2003), pp. 118–127.

[15] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[16] Pfenning, F. and R. Davies, *A judgmental reconstruction of modal logic*, Mathematical Structures in Computer Science **11** (2001), pp. 511–540, notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[17] Röckl, C., D. Hirschkoff and S. Berghofer, *Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts*, in: F. Honsell and M. Miculan, editors, *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)* (2001), pp. 364–378.

[18] Watkins, K., I. Cervesato, F. Pfenning and D. Walker, *A concurrent logical framework I: Judgments and properties*, Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University (2002), revised May 2003.

[19] Watkins, K., I. Cervesato, F. Pfenning and D. Walker, *A concurrent logical framework: The propositional fragment*, in: *Types for Proofs and Programs*, Springer-Verlag LNCS, 2004 Selected papers from the *Third International Workshop* Torino, Italy, April 2003. To appear.

[20] Woo, T. and S. Lam, *A semantic model for authentication protocols*, in: *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy* (1993), pp. 178–194.

A π-calculus encoding summarized

Syntax.

$$
\begin{array}{rcl}
\mathsf{eff} &=& \{1\} : \mathsf{type}. \\
\mathsf{latent} &:& \mathsf{label} \to \mathsf{eff}.
\end{array}
\qquad
\begin{array}{l}
\ulcorner [L_1, \ldots, L_n] \urcorner = \\
\quad \{\mathsf{let}\ \{1\} = \mathsf{latent}\ \ulcorner L_1 \urcorner\ \mathsf{in}\ \ldots \\
\quad\ \ \mathsf{let}\ \{1\} = \mathsf{latent}\ \ulcorner L_n \urcorner\ \mathsf{in}\ 1\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{name} &:& \mathsf{tp}. \\
\mathsf{chan} &:& \mathsf{tp} \to (\mathsf{nm} \to \mathsf{eff}) \to \mathsf{tp}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner \mathsf{Name} \urcorner &=& \mathsf{name} \\
\ulcorner \mathsf{Ch}(x{:}\tau)e \urcorner &=& \mathsf{chan}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner e \urcorner)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{stop} &:& \mathsf{pr}. \\
\mathsf{par} &:& \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{repeat} &:& \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{new} &:& \mathsf{tp} \to (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{choose} &:& \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{out} &:& \mathsf{nm} \to \mathsf{nm} \to \mathsf{pr}. \\
\mathsf{inp} &:& \mathsf{nm} \to \mathsf{tp} \to \\
&& \quad (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr}. \\
\mathsf{begin} &:& \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}. \\
\mathsf{end} &:& \mathsf{label} \to \mathsf{pr} \to \mathsf{pr}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner \mathsf{stop} \urcorner &=& \mathsf{stop} \\
\ulcorner P \mid Q \urcorner &=& \mathsf{par}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{repeat}\ P \urcorner &=& \mathsf{repeat}\ \ulcorner P \urcorner \\
\ulcorner \mathsf{new}(x{:}\tau); P \urcorner &=& \mathsf{new}\ \ulcorner \tau \urcorner\ (\lambda x. \ulcorner P \urcorner) \\
\ulcorner \mathsf{choose}\ P\ Q \urcorner &=& \mathsf{choose}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{out}\ x\langle y \rangle \urcorner &=& \mathsf{out}\ x\ y \\
\ulcorner \mathsf{inp}\ x(y{:}\tau); P \urcorner &=& \mathsf{inp}\ x\ \ulcorner \tau \urcorner\ (\lambda y. \ulcorner P \urcorner) \\
&& \\
\ulcorner \mathsf{begin}\ L; P \urcorner &=& \mathsf{begin}\ \ulcorner L \urcorner\ \ulcorner P \urcorner \\
\ulcorner \mathsf{end}\ L; P \urcorner &=& \mathsf{end}\ \ulcorner L \urcorner\ \ulcorner P \urcorner
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{tnil} &:& \mathsf{tr}. \\
\mathsf{tint} &:& \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tbegin} &:& \mathsf{label} \to \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tend} &:& \mathsf{label} \to \mathsf{tr} \to \mathsf{tr}. \\
\mathsf{tgen} &:& (\mathsf{nm} \to \mathsf{tr}) \to \mathsf{tr}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner \varepsilon \urcorner &=& \mathsf{tnil} \\
\ulcorner \tau, s \urcorner &=& \mathsf{tint}\ \ulcorner s \urcorner \\
\ulcorner \mathsf{begin}\ L, s \urcorner &=& \mathsf{tbegin}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\ulcorner \mathsf{end}\ L, s \urcorner &=& \mathsf{tend}\ \ulcorner L \urcorner\ \ulcorner s \urcorner \\
\ulcorner \mathsf{gen}\ \langle x \rangle, s \urcorner &=& \mathsf{tgen}\ (\lambda x. \ulcorner s \urcorner)
\end{array}
$$

Dynamic semantics.

$$
\begin{array}{rcl}
\mathsf{ev\_stop} &:& \mathsf{run\ stop} \multimap \{1\}. \\
\mathsf{ev\_par} &:& \mathsf{run}\ (\mathsf{par}\ P\ Q) \multimap \{\mathsf{run}\ P \otimes \mathsf{run}\ Q\}. \\
\mathsf{ev\_repeat} &:& \mathsf{run}\ (\mathsf{repeat}\ P) \multimap \{!\mathsf{run}\ P\}. \\
\mathsf{ev\_new} &:& \mathsf{run}\ (\mathsf{new}\ \tau\ (\lambda u.\, P\ u)) \multimap \{\exists u{:}\mathsf{nm}.\, \mathsf{run}\ (P\ u)\}. \\
\mathsf{ev\_choose}_i &:& \mathsf{run}\ (\mathsf{choose}\ P_1\ P_2) \multimap \{\mathsf{run}\ P_i\}. \\
\mathsf{ev\_sync} &:& \mathsf{run}\ (\mathsf{out}\ X\ Y) \multimap \mathsf{run}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \multimap \{\mathsf{run}\ (P\ Y)\}. \\
\mathsf{ev\_begin} &:& \Pi L{:}\mathsf{label}.\, \mathsf{run}\ (\mathsf{begin}\ L\ P) \multimap \{\mathsf{run}\ P\}. \\
\mathsf{ev\_end} &:& \Pi L{:}\mathsf{label}.\, \mathsf{run}\ (\mathsf{end}\ L\ P) \multimap \{\mathsf{run}\ P\}.
\end{array}
$$

Static semantics.

$$
\begin{array}{rcl}
\mathsf{wflab} & : & \mathsf{label} \longrightarrow \mathsf{type}.
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{wfeff} & : & \mathsf{eff} \longrightarrow \mathsf{type}. \\
\mathsf{wff\_eps} & : & \mathsf{wfeff}\ \{1\}. \\
\mathsf{wff\_lat} & : & \mathsf{wfeff}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\} \longleftarrow \mathsf{wflab}\ L \longleftarrow \mathsf{wfeff}\ E.
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{wftp} & : & \mathsf{tp} \longrightarrow \mathsf{type}. \\
\mathsf{wf\_name} & : & \mathsf{wftp}\ \mathsf{name}. \\
\mathsf{wf\_chan} & : & \mathsf{wftp}\ (\mathsf{chan}\ \tau\ (\lambda x.\, E\ x)) \longleftarrow \mathsf{wftp}\ \tau \longleftarrow (\Pi x.\, \mathsf{has}\ x\ \tau \longrightarrow \mathsf{wfeff}\ (E\ x)).
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{consume} & : & \mathsf{eff} \longrightarrow \mathsf{type}. \\
\mathsf{assume} & : & \mathsf{eff} \longrightarrow \mathsf{pr} \longrightarrow \mathsf{type}.
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{con\_eps} & : & \mathsf{consume}\ \{1\} \multimapinv \top. \\
\mathsf{con\_join} & : & \mathsf{consume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\} \\
& & \quad \multimapinv \mathsf{effect}\ L \multimapinv \mathsf{consume}\ E.
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{ass\_eps} & : & \mathsf{assume}\ \{1\}\ P \multimapinv \mathsf{good}\ P. \\
\mathsf{ass\_join} & : & \mathsf{assume}\ \{\mathsf{let}\ \{1\} = \mathsf{latent}\ L\ \mathsf{in}\ \mathsf{let}\ \{1\} = E\ \mathsf{in}\ 1\} \\
& & \quad \multimapinv (\mathsf{effect}\ L \multimap \mathsf{assume}\ E\ P).
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{has} & : & \mathsf{nm} \longrightarrow \mathsf{tp} \longrightarrow \mathsf{type}. \\
\mathsf{good} & : & \mathsf{pr} \longrightarrow \mathsf{type}.
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{gd\_stop} & : & \mathsf{good}\ \mathsf{stop} \multimapinv \top. \\
\mathsf{gd\_par} & : & \mathsf{good}\ (\mathsf{par}\ P\ Q) \multimapinv \mathsf{good}\ P \multimapinv \mathsf{good}\ Q. \\
\mathsf{gd\_repeat} & : & \mathsf{good}\ (\mathsf{repeat}\ P) \multimapinv \top \longleftarrow \mathsf{good}\ P. \\
\mathsf{gd\_new} & : & \mathsf{good}\ (\mathsf{new}\ \tau\ (\lambda x.\, P\ x)) \longleftarrow \mathsf{wftp}\ \tau \\
& & \quad \multimapinv (\Pi x{:}\mathsf{nm}.\, \mathsf{has}\ x\ \tau \longrightarrow \mathsf{good}\ (P\ x)). \\
\mathsf{gd\_choose} & : & \mathsf{good}\ (\mathsf{choose}\ P\ Q) \multimapinv (\mathsf{good}\ P \mathbin{\&} \mathsf{good}\ Q). \\
\mathsf{gd\_out} & : & \mathsf{good}\ (\mathsf{out}\ X\ Y) \longleftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \longleftarrow \mathsf{has}\ Y\ \tau \\
& & \quad \multimapinv \mathsf{consume}\ (E\ Y). \\
\mathsf{gd\_inp} & : & \mathsf{good}\ (\mathsf{inp}\ X\ \tau\ (\lambda y.\, P\ y)) \longleftarrow \mathsf{has}\ X\ (\mathsf{chan}\ \tau\ (\lambda y.\, E\ y)) \\
& & \quad \longleftarrow (\Pi y{:}\mathsf{nm}.\, \mathsf{has}\ y\ \tau \longrightarrow \mathsf{assume}\ (E\ y)\ (P\ y)). \\
\mathsf{gd\_begin} & : & \mathsf{good}\ (\mathsf{begin}\ L\ P) \multimapinv (\mathsf{effect}\ L \multimap \mathsf{good}\ P). \\
\mathsf{gd\_end} & : & \mathsf{good}\ (\mathsf{end}\ L\ P) \multimapinv \mathsf{effect}\ L \multimapinv \mathsf{good}\ P.
\end{array}
$$

Abstraction.

$$
\begin{aligned}
\mathsf{abst} &: \{\top\} \to \mathsf{tr} \to \mathsf{type}.\\[4pt]
\mathsf{abst\_nil} &: \mathsf{abst}\ E\ \mathsf{tnil}.\\
\mathsf{abst\_stop} &: \mathsf{abst}\ \{\mathsf{let}\ \{1\} = \mathsf{ev\_stop}\,{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ \mathsf{in}\ \langle\rangle\}\ s \leftarrow \mathsf{abst}\ E\ s.\\
\mathsf{abst\_par} &: \mathsf{abst}\ \{\mathsf{let}\ \{r_1 \otimes r_2\} = \mathsf{ev\_alt}\,{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,{}^\wedge r_1{}^\wedge r_2\ \mathsf{in}\ \langle\rangle\}\ s\\
&\qquad \leftarrow (\Pi r_1.\, \Pi r_2.\, \mathsf{abst}\ (E\,{}^\wedge r_1{}^\wedge r_2)\ s).\\
\mathsf{abst\_repeat} &: \mathsf{abst}\ \{\mathsf{let}\ \{!r\} = \mathsf{ev\_repeat}\,{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ r\ \mathsf{in}\ \langle\rangle\}\ s\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s).\\
\mathsf{abst\_new} &: \mathsf{abst}\ \{\mathsf{let}\ \{[x, r]\} = \mathsf{ev\_new}\,{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\ x\,{}^\wedge r\ \mathsf{in}\ \langle\rangle\}\\
&\qquad\quad (\mathsf{tgen}\ (\lambda x.\, s\ x))\\
&\qquad \leftarrow (\Pi x.\, \Pi r.\, \mathsf{abst}\ (E\ x\,{}^\wedge r)\ (s\ x)).\\
\mathsf{abst\_choose}_i &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_choose}_i{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,{}^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\ r)\ s).\\
\mathsf{abst\_sync} &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_sync}\,{}^\wedge R_1{}^\wedge R_2\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,{}^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tint}\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\,{}^\wedge r)\ s).\\
\mathsf{abst\_begin} &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_begin}\ L\,{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,{}^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tbegin}\ L\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\,{}^\wedge r)\ s).\\
\mathsf{abst\_end} &: \mathsf{abst}\ \{\mathsf{let}\ \{r\} = \mathsf{ev\_end}\ L\,{}^\wedge R\ \mathsf{in}\ \mathsf{let}\ \{\_\} = E\,{}^\wedge r\ \mathsf{in}\ \langle\rangle\}\ (\mathsf{tend}\ L\ s)\\
&\qquad \leftarrow (\Pi r.\, \mathsf{abst}\ (E\,{}^\wedge r)\ s).
\end{aligned}
$$

Safety.

$$
\begin{aligned}
\mathsf{invalid} &: \mathsf{tr} \to \mathsf{type}.\\
\mathsf{remove} &: \mathsf{label} \to \mathsf{tr} \to \mathsf{tr} \to \mathsf{type}.\\
\neq &: \mathsf{label} \to \mathsf{label} \to \mathsf{type}.\\[4pt]
\mathsf{inval\_end} &: \mathsf{invalid}\ (\mathsf{tend}\ \_\ \_).\\
\mathsf{inval\_int} &: \mathsf{invalid}\ (\mathsf{tint}\ s) \leftarrow \mathsf{invalid}\ s.\\
\mathsf{inval\_gen} &: \mathsf{invalid}\ (\mathsf{tgen}\ (\lambda x.\, s\ x)) \leftarrow (\Pi x.\, \mathsf{invalid}\ (s\ x)).\\
\mathsf{inval\_begin} &: \mathsf{invalid}\ (\mathsf{tbegin}\ L\ s) \leftarrow \mathsf{remove}\ L\ s\ s' \leftarrow \mathsf{invalid}\ s'.\\[4pt]
\mathsf{rem\_match} &: \mathsf{remove}\ L\ (\mathsf{tend}\ L\ s)\ s.\\
\mathsf{rem\_nil} &: \mathsf{remove}\ L\ \mathsf{tnil}\ \mathsf{tnil}.\\
\mathsf{rem\_int} &: \mathsf{remove}\ L\ (\mathsf{tint}\ s)\ (\mathsf{tint}\ s') \leftarrow \mathsf{remove}\ L\ s\ s'.\\
\mathsf{rem\_gen} &: \mathsf{remove}\ L\ (\mathsf{tgen}\ (\lambda x.\, s\ x))\ (\mathsf{tgen}\ (\lambda x.\, s\ x))\\
&\qquad \leftarrow (\Pi x.\, \mathsf{remove}\ L\ (s\ x)\ (s'\ x)).\\
\mathsf{rem\_begin} &: \mathsf{remove}\ L\ (\mathsf{tbegin}\ L'\ s)\ (\mathsf{tbegin}\ L'\ s') \leftarrow \mathsf{remove}\ L\ s\ s'.\\
\mathsf{rem\_end} &: \mathsf{remove}\ L\ (\mathsf{tend}\ L'\ s)\ (\mathsf{tend}\ L'\ s') \leftarrow L \neq L' \leftarrow \mathsf{remove}\ s\ s'.\\[4pt]
\mathsf{invalid} &: \mathsf{tr} \to \mathsf{type}.\\
\mathsf{unsafe} &: \mathsf{pr} \to \mathsf{type}.\\[4pt]
\mathsf{show\_unsafe} &: \Pi E{:}(\mathsf{run}\ P \multimap \{\top\}).\, \mathsf{unsafe}\ P \leftarrow (\Pi r.\, \mathsf{abst}\ (E\,{}^\wedge r)\ s) \leftarrow \mathsf{invalid}\ s.
\end{aligned}
$$

B CLF type theory summarized

See the technical report [18] for further details.

Syntax.

$$
\begin{aligned}
K, L &::= \mathsf{type} \mid \Pi u\!:\!A.\, K \\[4pt]
A, B, C &::= A \multimap B \mid \Pi u\!:\!A.\, B \mid A \,\&\, B \\
&\quad\ \mid \top \mid \{S\} \mid P \\
P &::= a \mid P\ N \\
S &::= \exists u\!:\!A.\, S \mid S_1 \otimes S_2 \mid 1 \mid\, !A \mid A \\[4pt]
\Gamma &::= \cdot \mid \Gamma, u\!:\!A \\
\Delta &::= \cdot \mid \Delta, x \hat{:} A \\
\Sigma &::= \cdot \mid \Sigma, a\!:\!K \mid \Sigma, c\!:\!A \\
N &::= \hat{\lambda} x.\, N \mid \lambda u.\, N \mid \langle N_1, N_2 \rangle \\
&\quad\ \mid \langle\rangle \mid \{E\} \mid R \\
R &::= c \mid u \mid x \mid R \,\hat{}\, N \mid R\ N \mid \pi_1 R \mid \pi_2 R \\
E &::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\
M &::= [N, M] \mid M_1 \otimes M_2 \mid 1 \mid\, !N \mid N \\[4pt]
p &::= [u, p] \mid p_1 \otimes p_2 \mid 1 \mid\, !u \mid x \\[4pt]
\Psi &::= p \hat{:} S, \Psi \mid \cdot
\end{aligned}
$$

Typing.

Judgments.

$$
\begin{array}{ccc}
\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind} & \Gamma; \Delta \vdash_\Sigma N \Leftarrow A & \vdash \Sigma\ \mathsf{ok} \\
\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} & \Gamma; \Delta \vdash_\Sigma R \Rightarrow A & \vdash_\Sigma \Gamma\ \mathsf{ok} \\
\Gamma \vdash_\Sigma P \Rightarrow K & \Gamma; \Delta \vdash_\Sigma E \leftarrow S & \Gamma \vdash_\Sigma \Delta\ \mathsf{ok} \\
\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type} & \Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S & \Gamma \vdash_\Sigma \Psi\ \mathsf{ok} \\
 & \Gamma; \Delta \vdash_\Sigma M \Leftarrow S &
\end{array}
$$

$$
\begin{aligned}
\mathsf{inst\_k}_A(u.\, K, N) &= K' \\
\mathsf{inst\_a}_A(u.\, B, N) &= B' \\
\mathsf{inst\_s}_A(u.\, S, N) &= S'
\end{aligned}
$$

Rules.

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \qquad \frac{\vdash \Sigma\ \mathsf{ok} \quad \cdot \vdash_\Sigma K \Leftarrow \mathsf{kind}}{\vdash \Sigma, a{:}K\ \mathsf{ok}} \qquad \frac{\vdash \Sigma\ \mathsf{ok} \quad \cdot \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash \Sigma, c{:}A\ \mathsf{ok}}$$

$$\frac{}{\vdash_\Sigma \cdot\ \mathsf{ok}} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ok} \quad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash_\Sigma \Gamma, u{:}A\ \mathsf{ok}}$$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ \mathsf{ok}} \qquad \frac{\Gamma \vdash_\Sigma \Delta\ \mathsf{ok} \quad \Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}}{\Gamma \vdash_\Sigma \Delta, x \hat{:} A\ \mathsf{ok}}$$

$$\frac{}{\Gamma \vdash_\Sigma \cdot\ \mathsf{ok}} \qquad \frac{\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type} \quad \Gamma \vdash_\Sigma \Psi\ \mathsf{ok}}{\Gamma \vdash_\Sigma p \hat{:} S, \Psi\ \mathsf{ok}}$$

Henceforth, it will be assumed that all judgments are considered relative to a particular fixed signature Σ, and the signature indexing each of the other typing judgments will be suppressed.

$$\frac{}{\Gamma \vdash \mathsf{type} \Leftarrow \mathsf{kind}}\ \mathsf{type}\mathbf{KF} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u{:}A \vdash K \Leftarrow \mathsf{kind}}{\Gamma \vdash \Pi u{:}A.\, K \Leftarrow \mathsf{kind}}\ \Pi\mathbf{KF}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \multimap B \Leftarrow \mathsf{type}}\ {\multimap}\mathbf{F} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u{:}A \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash \Pi u{:}A.\, B \Leftarrow \mathsf{type}}\ \Pi\mathbf{F}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \,\&\, B \Leftarrow \mathsf{type}}\ \&\mathbf{F} \qquad \frac{}{\Gamma \vdash \top \Leftarrow \mathsf{type}}\ \top\mathbf{F}$$

$$\frac{\Gamma \vdash S \Leftarrow \mathsf{type}}{\Gamma \vdash \{S\} \Leftarrow \mathsf{type}}\ \{\}\mathbf{F} \qquad \frac{\Gamma \vdash P \Rightarrow \mathsf{type}}{\Gamma \vdash P \Leftarrow \mathsf{type}}\ {\Rightarrow}\mathsf{type}{\Leftarrow}$$

$$\frac{}{\Gamma \vdash a \Rightarrow \Sigma(a)}\ a \qquad \frac{\Gamma \vdash P \Rightarrow \Pi u{:}A.\, K \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma \vdash P\ N \Rightarrow \mathsf{inst\_k}_A(u.\, K, N)}\ \Pi\mathbf{KE}$$

$$\frac{\Gamma \vdash S_1 \Leftarrow \mathsf{type} \quad \Gamma \vdash S_2 \Leftarrow \mathsf{type}}{\Gamma \vdash S_1 \otimes S_2 \Leftarrow \mathsf{type}}\ \otimes\mathbf{F} \qquad \frac{}{\Gamma \vdash 1 \Leftarrow \mathsf{type}}\ 1\mathbf{F}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, u{:}A \vdash S \Leftarrow \mathsf{type}}{\Gamma \vdash \exists u{:}A.\, S \Leftarrow \mathsf{type}}\ \exists\mathbf{F} \qquad \frac{\Gamma \vdash A \Leftarrow \mathsf{type}}{\Gamma \vdash {!}A \Leftarrow \mathsf{type}}\ {!}\mathbf{F}$$

$$\frac{\Gamma; \Delta, x \hat{:} A \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \hat{\lambda} x.\, N \Leftarrow A \multimap B}\ {\multimap}\mathbf{I} \qquad \frac{\Gamma, u{:}A; \Delta \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \lambda u.\, N \Leftarrow \Pi u{:}A.\, B}\ \Pi\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash N_1 \Leftarrow A \quad \Gamma; \Delta \vdash N_2 \Leftarrow B}{\Gamma; \Delta \vdash \langle N_1, N_2 \rangle \Leftarrow A \,\&\, B}\ \&\mathbf{I} \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle \Leftarrow \top}\ \top\mathbf{I}$$

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta \vdash \{E\} \Leftarrow \{S\}}\ \{\}\mathbf{I} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow P' \quad P' \equiv P}{\Gamma; \Delta \vdash R \Leftarrow P}\ {\Rightarrow}{\Leftarrow}$$

$$\frac{}{\Gamma; \cdot \vdash c \Rightarrow \Sigma(c)}\ c \qquad \frac{}{\Gamma; \cdot \vdash u \Rightarrow \Gamma(u)}\ u \qquad \frac{}{\Gamma; x \mathbin{\hat{:}} A \vdash x \Rightarrow A}\ x$$

$$\frac{\Gamma; \Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma; \Delta_2 \vdash N \Leftarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash R\,\hat{}\,N \Rightarrow B} \multimap\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow A \mathbin{\&} B}{\Gamma; \Delta \vdash \pi_1 R \Rightarrow A} \mathbin{\&}\mathbf{E}_1$$

$$\frac{\Gamma; \Delta \vdash R \Rightarrow \Pi u\!:\!A.\, B \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \Delta \vdash R\ N \Rightarrow \mathsf{inst\_a}_A(u.\, B, N)} \Pi\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow A \mathbin{\&} B}{\Gamma; \Delta \vdash \pi_2 R \Rightarrow B} \mathbin{\&}\mathbf{E}_2$$

$$\frac{\Gamma; \Delta_1 \vdash R \Rightarrow \{S_0\} \quad \Gamma; \Delta_2; p \mathbin{\hat{:}} S_0 \vdash E \leftarrow S}{\Gamma; \Delta_1, \Delta_2 \vdash (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) \leftarrow S} \{\}\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash M \Leftarrow S}{\Gamma; \Delta \vdash M \leftarrow S} \Leftarrow\leftarrow$$

$$\frac{\Gamma; \Delta; p_1 \mathbin{\hat{:}} S_1, p_2 \mathbin{\hat{:}} S_2, \Psi \vdash E \leftarrow S}{\Gamma; \Delta; p_1 \otimes p_2 \mathbin{\hat{:}} S_1 \otimes S_2, \Psi \vdash E \leftarrow S} \otimes\mathbf{L} \qquad \frac{\Gamma; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; 1 \mathbin{\hat{:}} 1, \Psi \vdash E \leftarrow S} 1\mathbf{L}$$

$$\frac{\Gamma, u\!:\!A; \Delta; p \mathbin{\hat{:}} S_0, \Psi \vdash E \leftarrow S}{\Gamma; \Delta; [u, p] \mathbin{\hat{:}} \exists u\!:\!A.\, S_0, \Psi \vdash E \leftarrow S} \exists\mathbf{L} \qquad \frac{\Gamma, u\!:\!A; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; !u \mathbin{\hat{:}} !A, \Psi \vdash E \leftarrow S} !\mathbf{L}$$

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta; \cdot \vdash E \leftarrow S} \leftarrow\leftarrow \qquad \frac{\Gamma; \Delta, x \mathbin{\hat{:}} A; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; x \mathbin{\hat{:}} A, \Psi \vdash E \leftarrow S} \mathrm{A}\mathbf{L}$$

$$\frac{\Gamma; \Delta_1 \vdash M_1 \Leftarrow S_1 \quad \Gamma; \Delta_2 \vdash M_2 \Leftarrow S_2}{\Gamma; \Delta_1, \Delta_2 \vdash M_1 \otimes M_2 \Leftarrow S_1 \otimes S_2} \otimes\mathbf{I} \qquad \frac{}{\Gamma; \cdot \vdash 1 \Leftarrow 1} 1\mathbf{I}$$

$$\frac{\Gamma; \cdot \vdash N \Leftarrow A \quad \Gamma; \Delta \vdash M \Leftarrow \mathsf{inst\_s}_A(u.\, S, N)}{\Gamma; \Delta \vdash [N, M] \Leftarrow \exists u\!:\!A.\, S} \exists\mathbf{I} \qquad \frac{\Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \cdot \vdash !N \Leftarrow !A} !\mathbf{I}$$