A Concurrent Logical Framework II: Examples and Applications

Iliano Cervesato   Frank Pfenning   David Walker
Kevin Watkins

March 2002, revised May 2003
CMU-CS-02-102

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the National Science Foundation under the title "Meta-Logical Frameworks," grants CCR-9619584 and CCR-9988281; and in part by NRL under grant N00173-00-C-2086. Iliano Cervesato was partially supported by NRL under contract N00173-00-C-2086 and by NSF grant INT98-15731. David Walker was partially supported by the Office of Naval Research under the title "Efficient Logics for Network Security," grant N00014-01-1-0432; by DARPA under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," Contract No. F30603-99-1-0519; and by Microsoft Research. A part of this work was completed while David Walker was a post-doctoral fellow at Carnegie Mellon University.

Abstract

CLF is a new logical framework with an intrinsic notion of concurrency. It is designed as a conservative extension of the linear logical framework LLF with the synchronous connectives $\otimes$, 1, $!$, and $\exists$ of intuitionistic linear logic, encapsulated in a monad. LLF is itself a conservative extension of LF with the asynchronous connectives $\multimap$, $\&$ and $\top$.
In this report, the second of two technical reports describing CLF, we illustrate the expressive power of the framework by encoding several different concurrent languages including both the synchronous and asynchronous π-calculus, an ML-like language with futures, lazy evaluation and concurrency primitives in the style of CML, Petri nets and finally, the security protocol specification language MSR.
Throughout the report we assume the reader is already familiar with the formal definition of CLF. For detailed explanation and development of the type theory, please see *A Concurrent Logical Framework I: Judgments and Properties* [WCPW02].

Keywords: logical frameworks, type theory, linear logic, concurrency

Contents

List of Figures

1 Introduction

A logical framework [Pfe01b, BM01] 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 and uniform as possible, yet provide intrinsic means for representing common concepts and operations in its application domain.

The particular lineage of logical frameworks we are concerned with in this paper started with the Automath languages [dB80] which originated the use of dependent types. It was followed by LF [HHP93], crystallizing the *judgments-as-types* principle. LF is based on a minimal type theory λ^{Π} with only the dependent function type constructor Π. It nonetheless directly supports concise and elegant expression of variable renaming and capture-avoiding substitution at the level of syntax, and parametric and hypothetical judgments in deductions. Moreover, proofs are reified as objects which allows properties of or relations between proofs to be expressed within the framework [Pfe91].

Representations of systems involving state remained cumbersome until the design of the linear logical framework LLF [CP98] and its close relative RLF [IP98]. For example, LLF allows an elegant representation of Mini-ML with mutable references that reifies imperative computations as objects. 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 $\top$. This type theory corresponds to the largest freely generated fragment of intuitionistic linear logic [HM94, Bar96] whose proofs admit long normal forms without any commuting conversions. This allows a relatively simple type-directed equality-checking algorithm which is critical in the proof of decidability of type-checking for the framework [CP98, VC00].

While LLF solved many problems associated with stateful computation, the encoding of *concurrent computations* remained unsatisfactory. In this report, we demonstrate that the limitations of LLF can be overcome by extending the framework with a monad that incorporates the synchronous connectives $\otimes$, 1, $!$, and $\exists$ of intuitionistic linear logic. We call this new framework Concurrent LF (CLF).

Readers interested in the meta-theory of CLF should read the precursor to this report [WCPW02], which explains the formulation of the framework and describes its typing judgments and properties in detail. Here, we review the syntax of CLF and state some fundamental properties of the framework (Section 2). However, we give no explanation of the typing rules. They are merely included as a reference (see Appendix A).

The purpose of this report is to demonstrate the expressive power of CLF through a series of examples and, in particular, to focus on CLF's effectiveness at encoding concurrent programming paradigms. In Section 3, we present the essence of concurrent programming, the π-calculus [Mil99]. We give encodings of both the synchronous and asynchronous π-calculus and a proof of adequacy in the synchronous case. The adequacy proof for the asynchronous case follows similar, but simpler lines.

In Section 4, we give a novel encoding of an ML-like language with a destination-passing style operational semantics. The encoding is highly modular and we are easily able to treat a significant fragment of a practical programming language. More specifically, we show how to encode functions, recursion, definitions, unit type, pair type, mutable references, lazy evaluation, futures in the style of Multi-Lisp, and concurrency primitives in the style of

CML. Although we do not show it here, this encoding may easily be extended to include polymorphism, unit types, sum types, union types and intersection types.

In section 5, we demonstrate our framework's capacity for representing *truly concurrent* computations [Maz95] by giving an encoding of Petri nets and proving it adequate. We discuss both the operational semantics that identifies independent interleavings of transition applications and the description of the behavior of a Petri net stemming from trace theory.

In section 6, we explore a promising application area, the specification and verification of security protocols. In particular, we show how to encode MSR, a rich, strongly typed framework for representing cryptographic protocols. Finally, we conclude with section 7.

2 The Concurrent Logical Framework CLF

In contrast to prior presentations of the logical framework LF, all terms are represented in β-normal, η-long form—what in [HP00] are called quasi-canonical forms. The strategy based entirely on canonical forms also simplifies adequacy proofs for representations of other theories within CLF because such representations are always defined in terms of canonical forms.

This new presentation also simplifies the proof that type checking is decidable. Normally, the proof of decidability of type checking is quite involved because the type checking algorithm must compare the objects that appear in types for equality. However, in our framework, where all terms are β-normal, η-long, equality checking reduces to α-convertibility and is trivially decidable. Type checking is slightly more complex because checking dependent functions requires that we substitute terms into other types and terms. In order to maintain the invariant that all terms are β-normal, η-long substitution must simultaneously normalize objects. We call this new form of substitution *canonical substitution* and the reader is encouraged to examine the first technical report in this series for details[WCPW02].

2.1 Syntax

The syntactic presentation of canonical forms is based on a distinction between *normal objects* N and *atomic objects* R. It is convenient to make a similar distinction between *normal types* A and *atomic types* P, and to segregate the connectives restricted to the monad as the *monadic (synchronous) types* S. For kinds, there are no constants—there is only the symbol type—so there are only *normal kinds* K. A normal object is a series of introduction rules applied to atomic objects, while an atomic object is a series of natural-deduction style elimination rules applied to a variable or constant. The only elimination not permitted is the monad elimination rule, which is foreign to natural deduction.

In order to control the monad elimination rule, it is separated into a separate syntactic class of *expressions* E, only permitted directly inside a monad introduction. Introductions for the connectives restricted to the monad must occur immediately before the transition from objects to expressions. While this is already guaranteed by the syntactic restrictions on synchronous types, it is convenient to make the distinction at the level of the object syntax as well, so there is a class of *monadic (normal) objects* M. Eliminations of the connectives restricted to the monad are all invertible and are represented syntactically by *patterns* p.

We also use the symbol kind to classify the valid kinds. Throughout the presentation, terms that differ only in the names of their bound variables are considered to be the same. In addition, the metavariable Δ always denotes an equivalence class of linear contexts up to rearrangement.

$$
\begin{aligned}
K, L &::= \mathsf{type} \mid \Pi u{:}A.\, K \\
A, B, C &::= A \multimap B \mid \Pi u{:}A.\, B \mid A \,\&\, B \mid \top \mid \\
&\qquad \{S\} \mid P \\
P &::= a \mid P\, N \\
S &::= S_1 \otimes S_2 \mid 1 \mid \exists u{:}A.\, S \mid !A \mid A \\
\Gamma &::= \cdot \mid \Gamma, u{:}A \\
\Delta &::= \cdot \mid \Delta, x \hat{:} A \\
\Sigma &::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A
\end{aligned}
$$

$$
\begin{aligned}
N &::= \hat{\lambda} x.\, N \mid \lambda u.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R \\
R &::= c \mid u \mid x \mid R\,\hat{}\,N \mid R\; N \mid \pi_1 R \mid \pi_2 R \\
E &::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\
M &::= M_1 \otimes M_2 \mid 1 \mid [N, M] \mid !N \mid N \\
p &::= p_1 \otimes p_2 \mid 1 \mid [u, p] \mid !u \mid x \\
\Psi &::= \cdot \mid p \hat{:} S, \Psi
\end{aligned}
$$

2.2 Typing Judgments

There is a typing judgement for each syntactic category, as well as well-formedness judgements for contexts Γ and signatures Σ. Each of these judgements is defined in a completely syntax-directed manner, so termination and decidability of typing is clear. For each of the *normal* syntactic categories the operational interpretation of the type-checking judgement is that a putative type is provided, and the judgement holds if the term can be typed with the given type. In particular, a normal term such as $\lambda x.\, x$ may have several different types. This stands in contrast to the typical presentation of LF, where type labels are used in abstractions to ensure that every term has a unique type. For the *atomic* syntactic categories the situation is different: the operational meaning of the typing judgement is that it defines a partial function from an atomic term (in a given context and signature) to its unique type.

In all cases the typing judgement is not taken to have any particular meaning unless the context and signature referred to in the judgement are valid. For the normal syntactic categories, the typing judgement is meaningless unless the type referred to in the judgement is valid as well. For the atomic syntactic categories, it will be proved that whenever a typing is derivable and the context and signature mentioned in the typing are valid, the type mentioned in the judgement is valid. The judgements are as follows.

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

For the complete set of typing rules associated with each judgment form, please see Appendix A. For an explanation and motivation for this type theory, please see the companion technical report [WCPW02].

2.3 Definitional Equality

The notion of definitional equality ($=$) for CLF is based on α-equivalence and the following schema for expressions:

$$\frac{E_1 =_c E_2}{E_1 = E_2} \qquad \frac{M_1 = M_2}{M_1 =_c M_2} \qquad \frac{R_1 = R_2 \qquad E_1 =_c \epsilon[E_2]}{(\mathsf{let}\ \{p\} = R_1\ \mathsf{in}\ E_1) =_c \epsilon[\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2]}$$

where, in the second rule, the concurrent context ϵ is a prefix of let's terminated by a hole $\_$ (formally $\epsilon ::= \_ \mid \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \epsilon$). This rule is subject to the following side-conditions:

1. no variable bound by p is free in the conclusion;
2. no variable bound by p is bound by the context ϵ;
3. no variable free in R_2 is bound by the context ϵ.

The other rules for $=$ define congruences other all the other syntactic classes of CLF.

Definitional equality is a congruence over all syntactic classes of CLF, including $=_c$ over expressions. In particular it is reflexive, symmetric and transitive. These properties are shown to hold in [WCPW02].

2.4 Properties of CLF

In this section, we state some basic properties of CLF that will be needed in the proofs of adequacy of some of our encodings. More specifically, our encodings will often use the CLF context to represent the state of a concurrent computation. These contexts have standard structural properties that make this representation concise and elegant. Here, we use the notation $\Gamma \vdash J$ (and $\Gamma; \Delta \vdash J$) to denote any judgment that depends upon the context Γ (or $\Gamma; \Delta$). We denote the free variables of a type using the notation $FV(B)$.

Lemma 2.1 (Exchange).

- *If* $\Gamma, x{:}A, y{:}B, \Gamma' \vdash J$ *and* $x \notin FV(B)$ *then* $\Gamma, y{:}B, x{:}A, \Gamma' \vdash J$.
- *If* $\Gamma, x{:}A, y{:}B, \Gamma'; \Delta \vdash J$ *and* $x \notin FV(B)$ *then* $\Gamma, y{:}B, x{:}A, \Gamma'; \Delta \vdash J$.

Lemma 2.2 (Weakening).

- *If* $\Gamma \vdash J$ *then* $\Gamma, x{:}A \vdash J$.
- *If* $\Gamma; \Delta \vdash J$ *then* $\Gamma, x{:}A; \Delta \vdash J$.

Lemma 2.3 (Contraction).

- *If* $\Gamma, x{:}A, y{:}A \vdash J$ *then* $\Gamma, x{:}A \vdash J[x/y]$.
- *If* $\Gamma, x{:}A, y{:}A; \Delta \vdash J$ *then* $\Gamma, x{:}A; \Delta \vdash J[x/y]$.

Lemma 2.4 (Strengthening).

- *If* $\Gamma, x{:}A \vdash J$ *and* $x \notin FV(J)$ *then* $\Gamma \vdash J$.
- *If* $\Gamma, x{:}A; \Delta \vdash J$ *and* $x \notin FV(J) \bigcup FV(D)$ *then* $\Gamma; \Delta \vdash J$.

3 The π-Calculus in CLF

The π-calculus [Mil99] was created to capture the essence of concurrent programming just as the λ-calculus captures the essence of functional programming. Here we show how the syntax and operational semantics of two variants of the π-calculus can be captured in CLF. First, we give an encoding of the *asynchronous* π-calculus in which all of the operators are interpreted as logical connectives from CLF. Next, we show how to modify this encoding to deal with Milner's choice operator and the more complex communication primitives of the synchronous π-calculus. We give a proof of adequacy for the latter encoding.

Miller [Mil92] has also investigated the relationship between the π-calculus and linear logic. He interprets π-calculus expressions directly as logical operators in classical (multiple conclusion) linear logic as opposed to our intuitionistic logic. He gives three slightly different translations of the synchronous π-calculus, although none of them to correspond exactly to Milner's calculus. In contrast, we are able to represent Milner's calculus exactly (as well as a number of variants including the asynchronous π-calculus) and give a detailed proof of adequacy. On the other hand, we only encode the syntax and operational semantics of the π-calculus whereas Miller also investigates notions of observational equivalence in a restricted calculus in which no values are passed on channels.

The notation in this and following sections is somewhat abbreviated for readability. In particular, we omit outermost Π-quantifiers in constant declarations in a signature. These quantifiers can easily be reconstructed by an implementation of CLF along the lines of Twelf [PS99]. Whenever we omit the leading Π-quantifier in the type of a dependent function, we also omit the corresponding object at any application site for that function. Once again, these objects can be inferred by an implementation.

We also write $A \multimapinv B$ for $B \multimap A$ and $A \leftarrow B$ for $B \rightarrow A$ if we would like to emphasize the operational reading of a declaration: "*reduce the goal of proving A to the goal of proving B*" along the lines of Lolli [HM94] or LLF [CP98].

3.1 The Asynchronous π-calculus

3.1.1 Syntax

The asynchronous π-calculus has a simple syntax consisting of processes (P) and channels (u, v, w).

$$P, Q, R ::= \mathsf{0} \mid (P \mid Q) \mid \mathsf{new}\, u\, P \mid !P \mid u(v).P \mid \overline{u}\langle v \rangle$$

Channels are conduits that can be used to pass values between processes. In our simple language, the only values are channels themselves. A process may be an instruction to do nothing ($\mathsf{0}$) or an instruction to execute two processes P and Q in parallel ($P \mid Q$). The process $\mathsf{new}\, u\, P$ restricts the scope of the channel u to the process P and the process $!P$ acts as an unlimited number of copies of P. All communication occurs via input and output processes. The output process, $\overline{u}\langle v \rangle$, sends the message v on channel u. The corresponding input process, $u(w).P$, substitutes the output v for w in the body of P.

Representation Following the standard LF representation methodology [HHP93], we represent the π-calculus's two syntactic classes with two new CLF types.

$$
\begin{array}{lcl}
\mathsf{chan} & : & \mathsf{type}. \\
\mathsf{expr} & : & \mathsf{type}.
\end{array}
$$

Next, we represent the processes themselves as objects with type expr. Our representation will use higher-order abstract syntax [Pfe01b] to represent the bound variable v in the restriction $\mathsf{new}\, v\, P$ and in the input process $u(v).P$.

$$
\begin{array}{lcl}
\mathsf{0} & : & \mathsf{expr}. \\
\mathsf{par} & : & \mathsf{expr} \to \mathsf{expr} \to \mathsf{expr}. \\
\mathsf{new} & : & (\mathsf{chan} \to \mathsf{expr}) \to \mathsf{expr}. \\
! & : & \mathsf{expr} \to \mathsf{expr}. \\
\mathsf{out} & : & \mathsf{chan} \to \mathsf{chan} \to \mathsf{expr}. \\
\mathsf{in} & : & \mathsf{chan} \to (\mathsf{chan} \to \mathsf{expr}) \to \mathsf{expr}.
\end{array}
$$

The representation function $\ulcorner\cdot\urcorner$ maps π-calculus processes into CLF objects with type expr.

$$
\begin{aligned}
\ulcorner \mathsf{0} \urcorner &= \mathsf{0} \\
\ulcorner P \mid Q \urcorner &= \mathsf{par}\, \ulcorner P \urcorner\, \ulcorner Q \urcorner \\
\ulcorner \mathsf{new}\, u\, P \urcorner &= \mathsf{new}\ (\lambda u. \ulcorner P \urcorner) \\
\ulcorner !P \urcorner &= \mathsf{rep}\, \ulcorner P \urcorner \\
\ulcorner \overline{u}\langle v \rangle \urcorner &= \mathsf{out}\ u\ v \\
\ulcorner u(v).P \urcorner &= \mathsf{in}\ u\ (\lambda v. \ulcorner P \urcorner)
\end{aligned}
$$

3.1.2 Operational Semantics

The operational semantics of the π-calculus consists of two parts. First, a *structural congruence relation* divides the set of processes into congruence classes. The operational semantics does not distinguish between processes in the same class. For example, running the null process in parallel with P is equivalent to running P by itself. Similarly, $!P$ is equivalent to running arbitrarily many copies of P in parallel with $!P$. When P is equivalent to Q we write $P \equiv Q$. The complete rules for structural congruence are presented below.

$$
\frac{}{P \mid Q \equiv Q \mid P} \qquad \frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)} \qquad \frac{}{\mathsf{0} \mid P \equiv P}
$$

$$
\frac{}{\mathsf{0} \equiv \mathsf{new}\, u\, \mathsf{0}} \qquad \frac{}{P \mid (\mathsf{new}\, u\, Q) \equiv \mathsf{new}\, u\, (P \mid Q)}\,*
$$

$$
\frac{}{\mathsf{new}\, u\, (\mathsf{new}\, v\, P) \equiv \mathsf{new}\, v\, (\mathsf{new}\, u\, P)}
$$

$$
\frac{}{!P \equiv P \mid !P}
$$

$$
\frac{}{P \equiv P} \qquad \frac{P \equiv Q}{Q \equiv P} \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}
$$

$$
\frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad \frac{P \equiv P'}{\mathsf{new}\, u\, P \equiv \mathsf{new}\, u\, P'} \qquad \frac{P \equiv P'}{!P \equiv !P'}
$$

$$
\frac{P \equiv P'}{u(v).P \equiv u(v).P'}
$$

* The variable u is not free in P.

A second relation, $P \longrightarrow Q$, describes how actual computation occurs. There is one central rule that describes how input and output processes interact:

$$\frac{}{u(w).P \mid \overline{u}\langle v\rangle \longrightarrow [v/w]P}$$

The other rules that govern the relation $P \longrightarrow Q$ either accommodate the structural congruence relation or allow communication to occur within the scope of a new channel name or in parallel with an inactive process.

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

$$\frac{P \longrightarrow P'}{\mathsf{new}\, u\, P \longrightarrow \mathsf{new}\, u\, P'} \qquad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

We extend this single-step relation to a multi-step relation $P \longrightarrow^* Q$ through the following rules.

$$\frac{P \equiv Q}{P \longrightarrow^* Q} \qquad \frac{P \longrightarrow Q \quad Q \longrightarrow^* R}{P \longrightarrow^* R}$$

Representation At a coarse level a sequence of transitions in the π-calculus will be represented by a sequence of nested let-expressions in CLF, terminating in a unit element.

$$\Gamma; \Delta \vdash (\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \ldots \langle\rangle) \leftarrow \top$$

Here Γ contains declarations for channels $u{:}\mathsf{chan}$ and replicable processes $r{:}\mathsf{proc}\ P$, while Δ contains $x \hat{:} \mathsf{proc}\ Q$ for all other processes active in the initial state. The goal type $\top$ allows the computation to stop at any point, modeling the multi-step reaction relation. The computation steps consist of the atomic object R_1 consuming part of Δ and replacing it with new variables via the pattern p_1, and so on.

More precisely, the CLF expressions consist of two alternating phases. In the first phase, *expression decomposition*, a sequence of CLF let-expressions will decompose a π-calculus expression into CLF connectives. For example, the null process $\mathsf{0}$ will be interpreted as the CLF connective 1 and the parallel composition $\mid$ will be interpreted as $\otimes$. The decomposed fragments accumulate in the contexts.

In the second phase, the CLF connectives interact according to the rules of linear logic. The decomposition is such that the resulting contexts naturally obey the correct structural congruences, at least at the shallow level. For example, the implicit associativity and commutativity of contexts mirrors the structural congruences associated with parallel composition.

To make these ideas precise in the framework, we begin by defining a type family for undecomposed π-calculus expressions.

$$\mathsf{proc} \ : \ \mathsf{expr} \rightarrow \mathsf{type}.$$

Next, we specify rules for interpreting the various π-calculus instructions. For example, the exit rule interprets the null process as the multiplicative unit 1.

$$\mathsf{exit} \quad : \quad \mathsf{proc}\ 0 \multimap \{1\}.$$

The following derivation shows the action of this rule.

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma;\, \mathsf{x} \mathbin{\hat{:}} \mathsf{proc}\ 0 \vdash \mathsf{exit}^\wedge\mathsf{x} \Rightarrow \{1\}\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ \Gamma;\, \Delta \vdash E \leftarrow \top\end{array}}{\Gamma;\, \Delta;\, \cdot \vdash E \leftarrow \top}}{\Gamma;\, \Delta;\, 1 \mathbin{\hat{:}} 1 \vdash E \leftarrow \top}}{\Gamma;\, \Delta, \mathsf{x} \mathbin{\hat{:}} \mathsf{proc}\ 0 \vdash \mathsf{let}\ \{1\} = \mathsf{exit}^\wedge\mathsf{x}\ \mathsf{in}\ E \leftarrow \top}$$

Notice that the pattern $1 \mathbin{\hat{:}} 1$ simply disappears. Hence, the null process has no effect, as it should.

$$\mathsf{fork} \quad : \quad \mathsf{proc}\ (\mathsf{par}\ \mathsf{P}\ \mathsf{Q}) \multimap \{\mathsf{proc}\ \mathsf{P} \otimes \mathsf{proc}\ \mathsf{Q}\}.$$

The fork rule interprets parallel composition as the tensor product of the two processes. When the tensor pattern is eliminated, two assumptions $x \mathbin{\hat{:}} \mathsf{proc}\ P$, $y \mathbin{\hat{:}} \mathsf{proc}\ Q$ will remain in the context. Since CLF is insensitive to the order in which these assumptions appear in the context, either P or Q may be interpreted next. We may, therefore, think of P and Q as two concurrent processes waiting in parallel in the CLF context.

We interpret the process $\mathsf{new}\, u\, P$ using an existential type.

$$\mathsf{name} \quad : \quad \mathsf{proc}\ (\mathsf{new}\ (\lambda \mathsf{u}.\, \mathsf{P}\ \mathsf{u})) \multimap \{\exists \mathsf{u}{:}\mathsf{chan}.\, \mathsf{proc}\ (\mathsf{P}\ \mathsf{u})\}.$$

The elimination rule for existentials guarantees that a fresh name u will be added to the CLF context when it is eliminated and hence u may only appear in P, as is required by the π-calculus scope restriction operator.

The process rep P acts as arbitrarily many copies of P in parallel. We model this behavior using the exponential connective.

$$\mathsf{promote} \quad : \quad \mathsf{proc}\ (\mathsf{rep}\ \mathsf{P}) \multimap \{!(\mathsf{proc}\ \mathsf{P})\}.$$

Finally, we come to the communication primitives. Output is straightforward: It asynchronously places its message on the network using an auxiliary predicate. An input $u(w).P$, on the other hand, is more complex. We interpret it as a linear function that consumes network messages v on channel u and produces a new undecomposed process P with v replacing w.

$$\mathsf{msg} \quad : \quad \mathsf{chan} \rightarrow \mathsf{chan} \rightarrow \mathsf{type}.$$

$$\begin{array}{lcl}\mathsf{outp} & : & \mathsf{proc}\ (\mathsf{out}\ \mathsf{U}\ \mathsf{V}) \multimap \{\mathsf{msg}\ \mathsf{U}\ \mathsf{V}\}.\\ \mathsf{inp} & : & \mathsf{proc}\ (\mathsf{in}\ \mathsf{U}\ (\lambda \mathsf{w}.\, \mathsf{P}\ \mathsf{w})) \multimap \{\Pi \mathsf{v}{:}\mathsf{chan}.\, \mathsf{msg}\ \mathsf{U}\ \mathsf{v} \multimap \{\mathsf{proc}\ (\mathsf{P}\ \mathsf{v})\}\}.\end{array}$$

3.1.3 Alternate Encodings

Wherever possible we have tried to encode π-calculus processes using the CLF logical operators directly. This approach demonstrates the power of our framework and elucidates the connection between the asynchronous π-calculus and linear logic. However, in several cases there are other possible choices. For example, we might have left $\mathsf{proc}\ (\mathsf{rep}\ P)$ as an uninterpreted linear assumption and added an explicit copy rule to mirror process replication:

$$\mathsf{copy} \;:\; \mathsf{proc}\ (\mathsf{rep}\ \mathsf{P}) \multimap \{\mathsf{proc}\ (\mathsf{rep}\ \mathsf{P}) \otimes \mathsf{proc}\ \mathsf{P}\}.$$

We also might have chosen to leave both input and output processes uninterpreted and to add an explicit rule for reaction.

$$\mathsf{react} \;:\; \mathsf{proc}\ (\mathsf{in}\ \mathsf{U}\ (\lambda \mathsf{w}.\, \mathsf{P}\ \mathsf{w})) \multimap \mathsf{proc}\ (\mathsf{out}\ \mathsf{U}\ \mathsf{V}) \multimap \{\mathsf{proc}\ (\mathsf{P}\ \mathsf{V})\}.$$

A Non-Encoding In our first attempt to encode the π-calculus, the rules for interpreting π-calculus expressions took a slightly different form. Each rule mapped computations to computations as follows:

$$\begin{array}{lll} \mathsf{exit}' & : & \{\mathsf{proc}\ 0\} \multimap \{1\}. \\ \mathsf{fork}' & : & \{\mathsf{proc}\ (\mathsf{par}\ \mathsf{P}\ \mathsf{Q})\} \multimap \{\mathsf{proc}\ \mathsf{P} \otimes \mathsf{proc}\ \mathsf{Q}\}. \\ \dots & & \end{array}$$

This representation leads to too many CLF computations, some of them nonsensical. For example, there are suddenly several different computations that interpret the process $0 \mid 0$ including

$$\begin{array}{l} \mathsf{x} \hat{:} \mathsf{proc}\ (\mathsf{par}\ 0\ 0) \vdash \mathsf{let}\ \{\mathsf{x}_1 \otimes \mathsf{x}_2\} = \mathsf{fork}'{}^\wedge\{\mathsf{x}\}\ \mathsf{in} \\ \qquad \mathsf{let}\ \{1\} = \mathsf{exit}'{}^\wedge\{\mathsf{x}_1\}\ \mathsf{in} \\ \qquad \mathsf{let}\ \{1\} = \mathsf{exit}'{}^\wedge\{\mathsf{x}_2\}\ \mathsf{in}\ \langle\rangle \leftarrow \top \end{array}$$

$$\begin{array}{l} \mathsf{x} \hat{:} \mathsf{proc}\ (\mathsf{par}\ 0\ 0) \vdash \mathsf{let}\ \{\mathsf{x}_1 \otimes \mathsf{x}_2\} = \mathsf{fork}'{}^\wedge\{\mathsf{x}\}\ \mathsf{in} \\ \qquad \mathsf{let}\ \{1\} = \mathsf{exit}'{}^\wedge\{\mathsf{let}\ \{1\} = \mathsf{exit}'{}^\wedge\{\mathsf{x}_2\}\ \mathsf{in}\ \mathsf{x}_1\}\ \mathsf{in}\ \langle\rangle \leftarrow \top \end{array}$$

In the second computation, the argument of exit' is itself a computation, independent of the mainline. It was unclear what such tree-like computations might mean or how they might relate to the operational semantics of the π-calculus. Consequently, we changed our encoding and took great care to use monadic encapsulation to control the structure of our proofs properly.

3.2 The Synchronous π-Calculus

The synchronous π-calculus adds two main features to the asynchronous π-calculus, synchronous communication and a nondeterministic choice operator. In our next encoding, we use the synchronous connectives of linear logic to represent parallel composition, replicated processes and name restriction in conjunction with the asynchronous connectives of linear logic to select and synchronize input and output processes given a series of nondeterministic choices.

3.2.1 Syntax

In the syntax of the synchronous π-calculus, we replace the input and output processes from the asynchronous calculus with a nondeterministic choice of processes M. The null process is now defined to be the empty choice. Each element of the nondeterministic choice (also called a sum) is an action which may be silent ($\tau.P$), an input, or an output. Notice that outputs ($\overline{u}\langle v\rangle.P$) are considered synchronous because they wait on u until they react with an input and only at that point proceed to execute the process P.

$$\begin{array}{rrl}
\text{Process Expressions} & P & ::= (P \mid Q) \mid \mathsf{new}\, u\, P \mid !P \mid M \\
\text{Sums} & M & ::= \mathsf{0} \mid c + M \\
\text{Actions} & \text{c} & ::= \tau.P \mid u(v).P \mid \overline{u}\langle v\rangle.P
\end{array}$$

Representation We represent the four syntactic classes of the synchronous π-calculus with four CLF types.

$$\begin{array}{lcl}
\mathsf{chan} & : & \mathsf{type}. \\
\mathsf{expr} & : & \mathsf{type}. \\
\mathsf{sum} & : & \mathsf{type}. \\
\mathsf{act} & : & \mathsf{type}.
\end{array}$$

The process expressions themselves are represented as objects of type expr, sums by objects of type sum and actions by objects of type act. As before, we represent every channel u by a corresponding unrestricted variable $u{:}\mathsf{chan}$ of the same name.

$$\begin{array}{lcl}
\mathsf{par} & : & \mathsf{expr} \rightarrow \mathsf{expr} \rightarrow \mathsf{expr}. \\
\mathsf{new} & : & (\mathsf{chan} \rightarrow \mathsf{expr}) \rightarrow \mathsf{expr}. \\
\mathsf{rep} & : & \mathsf{expr} \rightarrow \mathsf{expr}. \\
\mathsf{sync} & : & \mathsf{sum} \rightarrow \mathsf{expr}. \\
\\
\mathsf{null} & : & \mathsf{sum}. \\
\mathsf{alt} & : & \mathsf{act} \rightarrow \mathsf{sum} \rightarrow \mathsf{sum}. \\
\\
\mathsf{silent} & : & \mathsf{expr} \rightarrow \mathsf{act}. \\
\mathsf{in} & : & \mathsf{chan} \rightarrow (\mathsf{chan} \rightarrow \mathsf{expr}) \rightarrow \mathsf{act}. \\
\mathsf{out} & : & \mathsf{chan} \rightarrow \mathsf{chan} \rightarrow \mathsf{expr} \rightarrow \mathsf{act}.
\end{array}$$

The representation function $\ulcorner\cdot\urcorner$ maps process expressions into CLF objects with type expr, $\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$ maps sums into CLF objects with type sum, and $\ulcorner\!\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner\!\urcorner$ maps actions into CLF objects with type act. As in the previous encoding, it uses higher-order abstract syntax to represent the bound variable v in the restriction $\mathsf{new}\, v\, P$ and in the input process $u(v).P$.

$$
\begin{aligned}
\ulcorner P \mid Q \urcorner &= \mathsf{par}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\
\ulcorner \mathsf{new}\, u\, P \urcorner &= \mathsf{new}\ (\lambda u. \ulcorner P \urcorner) \\
\ulcorner !P \urcorner &= \mathsf{rep}\ \ulcorner P \urcorner \\
\ulcorner M \urcorner &= \mathsf{sync}\ \ulcorner\!\ulcorner M \urcorner\!\urcorner
\end{aligned}
$$

$$
\begin{aligned}
\ulcorner\!\ulcorner 0 \urcorner\!\urcorner &= \mathsf{null} \\
\ulcorner\!\ulcorner c + M \urcorner\!\urcorner &= \mathsf{alt}\ \ulcorner\!\ulcorner\!\ulcorner c \urcorner\!\urcorner\!\urcorner\ \ulcorner\!\ulcorner M \urcorner\!\urcorner
\end{aligned}
$$

$$
\begin{aligned}
\ulcorner\!\ulcorner\!\ulcorner \tau.P \urcorner\!\urcorner\!\urcorner &= \mathsf{silent}\ \ulcorner P \urcorner \\
\ulcorner\!\ulcorner\!\ulcorner u(v).P \urcorner\!\urcorner\!\urcorner &= \mathsf{in}\ u\ (\lambda v. \ulcorner P \urcorner) \\
\ulcorner\!\ulcorner\!\ulcorner \overline{u}\langle v \rangle.P \urcorner\!\urcorner\!\urcorner &= \mathsf{out}\ u\ v\ \ulcorner P \urcorner
\end{aligned}
$$

3.2.2 Operational Semantics

To define the structural congruence relation $P \equiv Q$ for the synchronous π-calculus, we adopt all the rules from the asynchronous calculus except for the rules governing input and output (whose form has changed) and augment them with the following rules for sums and actions:

$$
\frac{c \equiv c'}{c + M \equiv c' + M} \qquad \frac{M \equiv M'}{c + M \equiv c + M'} \qquad \frac{}{c_1 + c_2 + M \equiv c_2 + c_1 + M}
$$

$$
\frac{P \equiv P'}{\tau.P \equiv \tau.P'} \qquad \frac{P \equiv P'}{u(v).P \equiv u(v).P'} \qquad \frac{P \equiv P'}{\overline{u}\langle v \rangle.P \equiv \overline{u}\langle v \rangle.P'}
$$

The additional rules ensure that our relation is a congruence relation over sums and actions, and moreover that the order of elements in a sum is inconsequential.

With the addition of silent actions, there are now two non-trivial reaction rules. The first rule selects the action $\tau.P$ from a sum and continues with P, throwing away the unchosen elements of the sum. The second rule selects an input process $u(v).P$ from one sum and an output process $\overline{u}\langle w \rangle.Q$ from another. The two processes synchronize as the value w is passed from output to input and the computation continues with $[w/v]P \mid Q$.

$$
\frac{}{\tau.P + M \longrightarrow P} \qquad \frac{}{(u(v).P + M) \mid (\overline{u}\langle w \rangle.Q + N) \longrightarrow [w/v]P \mid Q}
$$

As before, the rules above may be used under a channel name binder or in parallel with another process. The relation $P \longrightarrow^* Q$ is the reflexive and transitive closure of the one-step reaction relation.

Representation Once again, we define a type family for undecomposed π-calculus expressions and interpret parallel composition, name restriction and replication as before.

$$
\begin{aligned}
&\mathsf{proc} &&: \mathsf{expr} \rightarrow \mathsf{type}. \\[4pt]
&\mathsf{fork} &&: \mathsf{proc}\ (\mathsf{par}\ \mathsf{P}\ \mathsf{Q}) \multimap \{\mathsf{proc}\ \mathsf{P} \otimes \mathsf{proc}\ \mathsf{Q}\}. \\
&\mathsf{name} &&: \mathsf{proc}\ (\mathsf{new}\ (\lambda \mathsf{u}.\ \mathsf{P}\ \mathsf{u})) \multimap \{\exists \mathsf{u}\!:\!\mathsf{chan}.\ \mathsf{proc}\ (\mathsf{P}\ \mathsf{u})\}. \\
&\mathsf{promote} &&: \mathsf{proc}\ (\mathsf{rep}\ \mathsf{P}) \multimap \{!(\mathsf{proc}\ \mathsf{P})\}.
\end{aligned}
$$

A sum represents a non-deterministic, possibly synchronized choice. We therefore introduce a new type family to represent a sum waiting to react, and a decomposition rule for the sync coercion.

$$\begin{array}{lll} \mathsf{choice} & : & \mathsf{sum} \rightarrow \mathsf{type}. \\ \mathsf{suspend} & : & \mathsf{proc}\ (\mathsf{sync}\ \mathsf{M}) \multimap \{\mathsf{choice}\ \mathsf{M}\}. \end{array}$$

The degenerate case of a sum is the null process. As before, it has no effect and it is interpreted as the multiplicative unit 1.

$$\mathsf{exit} \ : \ \mathsf{choice}\ \mathsf{null} \multimap \{1\}.$$

The more interesting reaction rules fall into two families. The first family non-deterministically selects a particular guarded process from a sum. It does not refer to the state and therefore is neither linear nor monadic. Intuitively, it employs don't-know non-determinism and could backtrack, unlike the other rules that are written with don't-care non-determinism in mind.

$$\begin{array}{lll} \mathsf{select} & : & \mathsf{sum} \rightarrow \mathsf{act} \rightarrow \mathsf{type}. \\ \mathsf{this} & : & \mathsf{select}\ (\mathsf{alt}\ \mathsf{C}\ \mathsf{M})\ \mathsf{C}. \\ \mathsf{next} & : & \mathsf{select}\ \mathsf{M}\ \mathsf{C} \rightarrow \mathsf{select}\ (\mathsf{alt}\ \mathsf{C}'\ \mathsf{M})\ \mathsf{C}. \end{array}$$

The second family selects the guarded processes to react and operates on them to perform the actual reaction step. For an silent action we simply select a guarded process with prefix τ from a suspended sum. For a communication, we select two matching guarded processes from two different suspended sums.

$$\begin{array}{lll} \mathsf{internal} & : & \mathsf{choice}\ \mathsf{M} \multimap \mathsf{select}\ \mathsf{M}\ (\mathsf{silent}\ \mathsf{P}) \rightarrow \{\mathsf{proc}\ \mathsf{P}\}. \\ \mathsf{external} & : & \mathsf{choice}\ \mathsf{M}_1 \multimap \mathsf{choice}\ \mathsf{M}_2 \multimap \\ & & \quad \mathsf{select}\ \mathsf{M}_1\ (\mathsf{in}\ \mathsf{U}\ (\lambda \mathsf{w}{:}\mathsf{chan}.\ \mathsf{P}\ \mathsf{w})) \rightarrow \mathsf{select}\ \mathsf{M}_2\ (\mathsf{out}\ \mathsf{U}\ \mathsf{V}\ \mathsf{Q}) \rightarrow \\ & & \quad \{\mathsf{proc}\ (\mathsf{par}\ (\mathsf{P}\ \mathsf{V})\ \mathsf{Q})\}. \end{array}$$

Note that substitution $[w/v]P$ in the reaction rule $(u(v).P + M) \mid (\overline{u}\langle w \rangle.Q + N) \longrightarrow [w/v]P \mid Q$ is accomplished by a corresponding substitution in the framework, which takes place when the canonical substitution of a process expression $\lambda v. \ldots$ for $P{:}\mathsf{chan} \rightarrow \mathsf{expr}$ is carried out. This is a minor variant of a standard technique in logical frameworks.

3.2.3 Adequacy

The representation of the syntax of the synchronous π-calculus uses higher-order abstract syntax and other well-known strategies from the standard LF representation methodology [HHP93]. Consequently, it should come as no surprise that our representation of the syntax of π-calculus is adequate.

Theorem 1 (Adequacy of Representation of Syntax). *Let* $\Gamma = u_1{:}\mathsf{chan}, \ldots, u_n{:}\mathsf{chan}$.

a) $\Gamma; \Delta \vdash N \Leftarrow \mathsf{expr}$ *iff* $N = \ulcorner P \urcorner$ *where* P *may contain* $u_1 \ldots u_n$.

b) $\Gamma; \Delta \vdash N \Leftarrow \mathsf{sum}$ *iff* $N = \ulcorner\!\ulcorner M \urcorner\!\urcorner$ *where* M *may contain* $u_1 \ldots u_n$.

c) $\Gamma; \Delta \vdash N \Leftarrow \mathsf{act}$ *iff* $N = \ulcorner\!\ulcorner c \urcorner\!\urcorner$ *where* c *may contain* $u_1 \ldots u_n$*.*

d) The representation function is a compositional bijection.

Proof. Standard techniques from LF representation methodology [Pfe01b]. □

On the other hand, our representation of concurrent computations employs a number of new techniques and the adequacy proof is somewhat more involved. Here, we sketch of the main components of the proof. The interested reader may examine Appendix B for further details.

We begin by defining the CLF contexts that may arise during a concurrent computation. Notice that the possibilities for contexts are really quite limited. They are limited to the natural representations of channels, replicated processes, unreplicated processes (which may be further decomposed) and sums waiting to react. If it were not for the monadic encapsulation, we would have to deal with the possibility that the context contains partial applications of curried rules such as internal and external. In order to deal with such partial applications, one would have to complicate the notion of adequacy (if indeed there is a notion of adequacy that makes sense) and consider many more possible cases during the proof. Here, and in the rest of this section, we collapse the two parts of the context Γ and Δ into a single context Γ for the sake of brevity. We are still able to distinguish unrestricted assumptions $u{:}A$ from linear assumptions $x \hat{:} A$ by their syntax. Hence, there is no fundamental change to the type theory.

Definition 2 (General Contexts Γ). $\Gamma ::= . \mid \Gamma, u{:}\mathsf{chan} \mid \Gamma, u{:}\mathsf{proc}\ P \mid \Gamma, x \hat{:} \mathsf{proc}\ P \mid \Gamma, x \hat{:} \mathsf{choice}\ M$

Next, we define a relation $P \longleftrightarrow \Gamma$ between processes P (modulo the structural congruence relation) and contexts Γ.

Definition 3 (Representation Relation). $P \longleftrightarrow \Gamma$ *if and only if* $\llcorner \Gamma \lrcorner = Q$ *and* $Q \equiv P$ *where* $\llcorner \Gamma \lrcorner$ *is the inverse of the representation function* $\ulcorner \cdot \urcorner$*.*

Informally, our adequacy theorem will state that related objects step to related objects. In other words, we will show that if $P \longleftrightarrow \Gamma$ then P steps to Q if and only if Γ steps to Γ' and $Q \longleftrightarrow \Gamma'$. However, in order for this statement to make any sense we will have to define an appropriate notion of "steps to" on contexts. Unfortunately, we cannot directly relate one CLF computation step to one step in the π-calculus since process decomposition steps (exit, fork, new, etc.) have no π-calculus analogue.

To handle this discrepancy, we will define three relations on CLF contexts. The first, $\Gamma, E \Longrightarrow_s \Gamma'$, models the structural congruence relation of the π-calculus. The computation E is a series of process decomposition steps that will break down the structure of the context Γ and produce context Γ'. The second relation, $\Gamma, E \Longrightarrow \Gamma'$, models the single-step reduction relation. Here, E is a series of process decomposition steps followed by a single reaction rule, either internal or external. It reflects the fact that our CLF computations are arranged in two alternating phases. Finally, $\Gamma, E \Longrightarrow^\star \Gamma'$ is the reflexive and transitive closure of $\Gamma, E \Longrightarrow \Gamma'$. It models the multi-step reduction relation. All three relations make use of the following primitive notion of equality on CLF contexts.

Definition 4 (Context Equivalence). *Let assumptions of the form $x?A$ be either linear or unrestricted assumptions.*

$$\frac{}{\Gamma, x?A, y?B, \Gamma' \equiv \Gamma, y?B, x?A, \Gamma'}\; x \notin B, y \notin A \qquad \frac{}{\Gamma, \Gamma' \equiv \Gamma, u\!:\!\mathsf{chan}, \Gamma'}$$

$$\frac{}{\Gamma, u\!:\!\mathsf{chan}, \Gamma' \equiv \Gamma, \Gamma'}\; u \notin \Gamma, \Gamma' \qquad \frac{}{\Gamma \equiv \Gamma} \qquad \frac{\Gamma \equiv \Gamma'' \quad \Gamma'' \equiv \Gamma'}{\Gamma \equiv \Gamma'}$$

We will also employ the following notation for composing computations.

Definition 5 (Composition of Computations $E\langle E'\rangle$). *The composition of two computations E and E' with type $\top$, denoted $E\langle E'\rangle$, is the computation that results from substituting E' for the terminal $\langle\rangle$ in E.*

Definition 6 (Representation of structural equivalence $\Longrightarrow_s$). $\Gamma_1, E \Longrightarrow_s \Gamma_k$ *if*

1. $E = \langle\rangle$ *and* $\Gamma_1 \equiv \Gamma_k$, *or*
2. $E = \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle$ *and there is a normal derivation of the following form:*

$$\frac{\cdots \qquad \dfrac{\cdots}{\Gamma_2 \vdash E' \leftarrow \top}}{\Gamma_1 \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E' \leftarrow \top}$$

and $\Gamma_2, E' \Longrightarrow_s \Gamma_k$

and R is one of the following atomic forms (where we let x range over either linear or unrestricted variables):

$$\mathsf{exit}^\wedge x \qquad \mathsf{fork}^\wedge x \qquad \mathsf{name}^\wedge x \qquad \mathsf{promote}^\wedge x \qquad \mathsf{suspend}^\wedge x$$

Definition 7 (Representation of Single-Step Reduction $\Rightarrow$). $\Gamma_0, E\langle \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle\rangle \Rightarrow \Gamma_2'$ *iff* $\Gamma_0, E \Longrightarrow_s \Gamma_1$ *and*

$$\frac{\cdots \qquad \overline{\Gamma_2 \vdash \langle\rangle \leftarrow \top}}{\Gamma_1 \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle \leftarrow \top}$$

and $\Gamma_2 \equiv \Gamma_2'$ *and R is one of the following atomic forms (where we let x, x_1, x_2 range over either linear or unrestricted variables):*

$$\mathsf{external}^\wedge x\ N \qquad \mathsf{internal}^\wedge x_1{}^\wedge x_2\ N_1\ N_2$$

and $N, N_1, N_2 ::= \mathsf{this} \mid \mathsf{next} N$

Definition 8 (Representation of multi-step reduction $\Longrightarrow^\star$). $\Gamma_1, E \Longrightarrow^\star \Gamma_k$ *iff*

1. $\Gamma_1, E \Longrightarrow_s \Gamma_k$
2. $E = E_1\langle E_2\rangle$ *and* $\Gamma_1, E_1 \Rightarrow \Gamma_2$ *and* $\Gamma_2, E_2 \Longrightarrow^\star \Gamma_k$.

We are now in a position to state our adequacy results for computations.

Theorem 9 (Adequacy of Representation 1).

1. *If* $\Gamma, E \Longrightarrow_s \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \longleftrightarrow \Gamma'$*.*

2. *If* $\Gamma, E \Rightarrow \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \rightarrow Q$ *and* $Q \longleftrightarrow \Gamma'$*.*

3. *If* $\Gamma, E \Longrightarrow^\star \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \longrightarrow^* Q$ *and* $Q \longleftrightarrow \Gamma'$*.*

Theorem 10 (Adequacy of Representation 2).

1. *If* $P \equiv Q$ *and* $P \longleftrightarrow \Gamma$*, then* $Q \longleftrightarrow \Gamma$*.*

2. *If* $P \longleftrightarrow \Gamma$ *and* $P \rightarrow Q$*, then there exists* E *and* Γ' *such that* $\Gamma, E \Rightarrow \Gamma'$ *and* $Q \longleftrightarrow \Gamma'$*.*

3. *If* $P \longleftrightarrow \Gamma$ *and* $P \longrightarrow^* Q$*, then there exists* E *and* Γ' *such that* $\Gamma, E \Longrightarrow^\star \Gamma'$ *and* $Q \longleftrightarrow \Gamma'$*.*

The proof of Adequacy 1 is by induction on the structure of the computation E in each case. The proof of Adequacy 2 proceeds by induction on the process relation: on $P \equiv Q$ in part (*1*); induction on $P \longrightarrow Q$ in part (*2*); and induction on $P \longrightarrow^* Q$ in part (*3*). Please see Appendix B for details.

4 Concurrent ML in CLF

In this section we give a representation of Mini-ML in CLF with various advanced features. This encoding shows how the concurrency features can be used for a variety of purposes, including specifying a sequential semantics, lazy evaluation, and synchronous process communication in the style of Concurrent ML [Rep99]. It also shows how representations can exploit features of LF (dependent types) and LLF (asynchronous linear) in CLF, which adds synchronous linear connective to LLF.

4.1 Destination-Passing Style

Our formulation of Mini-ML distinguishes between expressions and values, which is convenient particularly when we come to the description concurrently. Furthermore, the representation is intrinsically typed in the sense that only Mini-ML expressions and values that are well-typed in Mini-ML will be well-typed in the framework. So we have a CLF type tp for Mini-ML types, and CLF types exp T and val T for Mini-ML expressions and values of type T, respectively. These type families are declared as follows.

$$\begin{array}{lll} \mathsf{tp} & : & \mathsf{type}. \\ \mathsf{exp} & : & \mathsf{tp} \rightarrow \mathsf{type}. \\ \mathsf{val} & : & \mathsf{tp} \rightarrow \mathsf{type}. \end{array}$$

The first novelty of our representation of evaluation is the pervasive use of *destination-passing style*. This is a convenient way to avoid the use of explicit continuations or evaluation contexts which simplifies the description of concurrency. It also makes the whole description more modular. So we have types dest T for destinations of type T. Note that initially there are no destinations; they are all created (as parameters) during evaluation. In that sense they are similar to locations in Mini-ML with references [CP98], although they are not necessarily imperative.

There are two basic type families to describe the operational semantics, $\mathsf{eval}\ E\ D$ which evaluates E with destination D, and $\mathsf{return}\ V\ D$ which returns the value V to destination D. The types of the expressions, values, and destinations must match. Therefore we declare:

$$\begin{array}{lcl}
\mathsf{dest} & : & \mathsf{tp} \rightarrow \mathsf{type}. \\
\mathsf{eval} & : & \mathsf{exp}\ T \rightarrow \mathsf{dest}\ T \rightarrow \mathsf{type}. \\
\mathsf{return} & : & \mathsf{val}\ T \rightarrow \mathsf{dest}\ T \rightarrow \mathsf{type}.
\end{array}$$

Without effects, the behavior of eval and return is as follows. If we have the linear assumption $\mathsf{e} \hat{:} \mathsf{eval}\ E\ D$ for a given expression E and destination D, then there is a computation to $\mathsf{r} \hat{:} \mathsf{return}\ V\ D$ if and only if the evaluation of E yields V. More generally, if we have some initial state represented as $\Gamma;\ \Delta$ then $\Gamma;\ \Delta, \mathsf{e} \hat{:} \mathsf{eval}\ E\ D$ computes to $\Gamma';\ \Delta', \mathsf{r} \hat{:} \mathsf{return}\ V\ D$, where Γ' and Δ' model the effects of the evaluation of E and other possibly concurrent computations.

$$\begin{array}{lcl}
\mathsf{evaluate} & : & \mathsf{exp}\ T \rightarrow \mathsf{val}\ T \rightarrow \mathsf{type}. \\
\mathsf{run} & : & \mathsf{evaluate}\ E\ V \\
& & \quad \circ\!\!- (\Pi \mathsf{d}{:}\mathsf{dest}\ T.\, \mathsf{eval}\ E\ \mathsf{d} \multimap \{\mathsf{return}\ \mathsf{d}\ V \otimes \top\}).
\end{array}$$

4.2 Sequential Functional Programming

We now introduce a number of concepts in a completely orthogonal manner. With few exceptions, they are organized around the corresponding types.

Values. In surface expressions, we only need to allow variables as values. This avoids ambiguity, since we do not need to decide if a given term should be considered a value or an expression when writing it down. However, we do not enforce this additional restriction—this could be accomplished by introducing a separate type $\mathsf{var}\ T$ for variables of type T. Instead, we allow every value as an expression. The corresponding coercion is value. Evaluating an expression $\mathsf{value}\ V$ returns the value V immediately.

$$\begin{array}{lcl}
\mathsf{value} & : & \mathsf{val}\ T \rightarrow \mathsf{exp}\ T. \\
\mathsf{ev_value} & : & \mathsf{eval}\ (\mathsf{value}\ V)\ D \multimap \{\mathsf{return}\ V\ D\}.
\end{array}$$

Recursion. Recursion introduces no new types and no new values. It is executed simply by unrolling the recursion. This means, for example, that $\mathsf{fix}(\lambda \mathsf{u}.\, \mathsf{u})$ is legal, but does not terminate.

$$\begin{array}{lcl}
\mathsf{fix} & : & (\mathsf{exp}\ T \rightarrow \mathsf{exp}\ T) \rightarrow \mathsf{exp}\ T. \\
\mathsf{ev_fix} & : & \mathsf{eval}\ (\mathsf{fix}\ (\lambda \mathsf{u}.\, E\ \mathsf{u}))\ D \multimap \{\mathsf{eval}\ (E\ (\mathsf{fix}\ (\lambda \mathsf{u}.\, E\ \mathsf{u})))\ D\}.
\end{array}$$

Definitions. Computation can be explicitly staged via definitions by let. Unlike ordinary ML, we do not tie properties of polymorphism to definitions, but prefer explicit type abstractions and applications. Inference or reconstruction is considered a property of the concrete syntax, and not the internal operational semantics of ML we are specifying here.

$$
\begin{array}{lll}
\mathsf{let} & : & \mathsf{exp}\ \mathrm{T} \rightarrow (\mathsf{val}\ \mathrm{T} \rightarrow \mathsf{exp}\ \mathrm{S}) \rightarrow \mathsf{exp}\ \mathrm{S}. \\
\mathsf{ev\_let} & : & \mathsf{eval}\ (\mathsf{let}\ \mathrm{E}_1\ (\lambda \mathsf{x}.\, \mathrm{E}_2\ \mathsf{x}))\ \mathrm{D} \\
 & & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
 & & \qquad \otimes (\Pi \mathrm{V}_1 {:} \mathsf{val}\ \mathrm{T}.\, \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \multimap \{\mathsf{eval}\ (\mathrm{E}_1\ \mathrm{V}_1)\ \mathrm{D}\})\}.
\end{array}
$$

Note that we use substitution instead of an explicit environment in order to bind a variable to a value. This does not lead to re-evaluation, since there is an explicit coercion from values to expressions. It would be straightforward to design a lower-level encoding where the bindings of variables to values are modeled through the use of destinations.

Also note how we use a linear assumption

$$\mathsf{e} \stackrel{\wedge}{:} \Pi \mathrm{V}_1 {:} \mathsf{val}\ \mathrm{T}.\, \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \multimap \{\mathsf{eval}\ (\mathrm{E}_2\ \mathrm{V}_1)\ \mathrm{D}\}$$

in order to sequence Mini-ML computation explicitly: the evaluation of the body E_2 of the mllet expression can not continue until the the expression E_1 has finished computing a value V_1.

The higher-order nature of the encoding could be avoided by introducing either a new kind of intermediate expression or type family. In that case the signature looks more flat, in the style of an abstract machine.

Natural Numbers. There is nothing particularly surprising about the representation. We introduce both new expressions and new values. During evaluation of the case construct we have to choose one branch or the other, but never both. This is represented naturally by the use of an additive conjunction & in the encoding.

$$
\begin{array}{lll}
\mathsf{nat} & : & \mathsf{tp}. \\
\mathsf{z} & : & \mathsf{exp\ nat}. \\
\mathsf{s} & : & \mathsf{exp\ nat} \rightarrow \mathsf{exp\ nat}. \\
\mathsf{case} & : & \mathsf{exp\ nat} \rightarrow \mathsf{exp}\ \mathrm{T} \rightarrow (\mathsf{val\ nat} \rightarrow \mathsf{exp}\ \mathrm{T}) \rightarrow \mathsf{exp}\ \mathrm{T}. \\
 & & \\
\mathsf{z}' & : & \mathsf{val\ nat}. \\
\mathsf{s}' & : & \mathsf{val\ nat} \rightarrow \mathsf{val\ nat}. \\
 & & \\
\mathsf{ev\_z} & : & \mathsf{eval\ z}\ \mathrm{D} \multimap \{\mathsf{return}\ \mathsf{z}'\ \mathrm{D}\}. \\
\mathsf{ev\_s} & : & \mathsf{eval}\ (\mathsf{s}\ \mathrm{E}_1)\ \mathrm{D} \\
 & & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest\ nat}.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
 & & \qquad \otimes (\Pi \mathrm{V}_1 {:} \mathsf{val\ nat}.\, \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \multimap \{\mathsf{return}\ (\mathsf{s}'\ \mathrm{V}_1)\ \mathrm{D}\})\}. \\
\mathsf{ev\_case} & : & \mathsf{eval}\ (\mathsf{case}\ \mathrm{E}_1\ \mathrm{E}_2\ (\lambda \mathsf{x}.\, \mathrm{E}_3\ \mathsf{x}))\ \mathrm{D} \\
 & & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest\ nat}.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
 & & \qquad \otimes ((\mathsf{return}\ \mathsf{z}'\ \mathsf{d}_1 \multimap \{\mathsf{eval}\ \mathrm{E}_2\ \mathrm{D}\}) \\
 & & \qquad\quad \&\ (\Pi \mathrm{V}_1' {:} \mathsf{exp\ nat}.\, \mathsf{return}\ (\mathsf{s}'\ \mathrm{V}_1')\ \mathsf{d}_1 \multimap \{\mathsf{eval}\ (\mathrm{E}_3\ \mathrm{V}_1')\ \mathrm{D}\}))\}.
\end{array}
$$

Functions. Again, the use of higher-order abstract syntax and substitution makes this quite simple. As before, we specify explicitly that the function has to be evaluated before the argument in an application. Concurrency or parallelism will be introduced explicitly later.

$$
\begin{array}{lcl}
\mathsf{arrow} & : & \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}. \\
\mathsf{lam} & : & (\mathsf{val}\ \mathrm{T}_2 \to \mathsf{exp}\ \mathrm{T}_1) \to \mathsf{exp}\ (\mathsf{arrow}\ \mathrm{T}_2\ \mathrm{T}_1). \\
\mathsf{app} & : & \mathsf{exp}\ (\mathsf{arrow}\ \mathrm{T}_2\ \mathrm{T}_1) \to \mathsf{exp}\ \mathrm{T}_2 \to \mathsf{exp}\ \mathrm{T}_1. \\
\\
\mathsf{lam}' & : & (\mathsf{val}\ \mathrm{T}_2 \to \mathsf{exp}\ \mathrm{T}_1) \to \mathsf{val}\ (\mathsf{arrow}\ \mathrm{T}_2\ \mathrm{T}_1). \\
\mathsf{ev\_lam} & : & \mathsf{eval}\ (\mathsf{lam}\ (\lambda \mathsf{x}.\, \mathrm{E}_1\ \mathsf{x}))\ \mathrm{D} \multimap \{\mathsf{return}\ (\mathsf{lam}'\ (\lambda \mathsf{x}.\, \mathrm{E}_1\ \mathsf{x}))\ \mathrm{D}\}. \\
\mathsf{ev\_app} & : & \mathsf{eval}\ (\mathsf{app}\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ (\mathsf{arrow}\ \mathrm{T}_2\ \mathrm{T}_1).\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{E}_1' {:} \mathsf{val}\ \mathrm{T}_2 \to \mathsf{exp}\ \mathrm{T}_1.\, \mathsf{return}\ (\mathsf{lam}\ (\lambda \mathsf{x}.\, \mathrm{E}_1'\ \mathsf{x}))\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\exists \mathsf{d}_2 {:} \mathsf{dest}\ \mathrm{T}_2.\, \mathsf{eval}\ \mathrm{E}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad \otimes (\Pi \mathrm{V}_2 {:} \mathsf{val}\ \mathrm{T}_2.\, \mathsf{return}\ \mathrm{V}_2\ \mathsf{d}_2 \multimap \{\mathsf{eval}\ (\mathrm{E}_1'\ \mathrm{V}_2)\ \mathrm{D}\})\})\}.
\end{array}
$$

Pairs. Are included here for completeness.

$$
\begin{array}{lcl}
\mathsf{cross} & : & \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}. \\
\mathsf{pair} & : & \mathsf{exp}\ \mathrm{T}_1 \to \mathsf{exp}\ \mathrm{T}_2 \to \mathsf{exp}\ (\mathsf{cross}\ \mathrm{T}_1\ \mathrm{T}_2). \\
\mathsf{fst} & : & \mathsf{exp}\ (\mathsf{cross}\ \mathrm{T}_1\ \mathrm{T}_2) \to \mathsf{exp}\ \mathrm{T}_1. \\
\mathsf{snd} & : & \mathsf{exp}\ (\mathsf{cross}\ \mathrm{T}_1\ \mathrm{T}_2) \to \mathsf{exp}\ \mathrm{T}_2. \\
\\
\mathsf{pair}' & : & \mathsf{val}\ \mathrm{T}_1 \to \mathsf{val}\ \mathrm{T}_2 \to \mathsf{val}\ (\mathsf{cross}\ \mathrm{T}_1\ \mathrm{T}_2). \\
\\
\mathsf{ev\_pair} & : & \mathsf{eval}\ (\mathsf{pair}\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ \mathrm{T}_1.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{V}_1 {:} \mathsf{val}\ \mathrm{T}_1.\, \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\exists \mathsf{d}_2 {:} \mathsf{dest}\ \mathrm{T}_2.\, \mathsf{eval}\ \mathrm{E}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad \otimes (\Pi \mathrm{V}_2 {:} \mathsf{val}\ \mathrm{T}_2.\, \mathsf{return}\ \mathrm{V}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad\quad \multimap \{\mathsf{return}\ (\mathsf{pair}'\ \mathrm{V}_1\ \mathrm{V}_2)\ \mathrm{D}\})\})\}. \\
\mathsf{ev\_fst} & : & \mathsf{eval}\ (\mathsf{fst}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ (\mathsf{cross}\ \mathrm{T}_1\ \mathrm{T}_2).\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{V}_1 {:} \mathsf{val}\ \mathrm{T}_1.\, \Pi \mathrm{V}_2 {:} \mathsf{val}\ \mathrm{T}_2.\, \mathsf{return}\ (\mathsf{pair}'\ \mathrm{V}_1\ \mathrm{V}_2)\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\mathsf{return}\ \mathrm{V}_1\ \mathrm{D}\})\}. \\
\mathsf{ev\_snd} & : & \mathsf{eval}\ (\mathsf{snd}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ (\mathsf{cross}\ \mathrm{T}_1\ \mathrm{T}_2).\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{V}_1 {:} \mathsf{val}\ \mathrm{T}_1.\, \Pi \mathrm{V}_2 {:} \mathsf{val}\ \mathrm{T}_2.\, \mathsf{return}\ (\mathsf{pair}'\ \mathrm{V}_1\ \mathrm{V}_2)\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\mathsf{return}\ \mathrm{V}_2\ \mathrm{D}\})\}.
\end{array}
$$

Unit. A unit type with one element is included here since some expressions with effects in ML return a unit value.

$$
\begin{array}{lcl}
\mathsf{one} & : & \mathsf{tp}. \\
\mathsf{unit} & : & \mathsf{exp\ one}. \\
\mathsf{unit}' & : & \mathsf{val\ one}. \\
\mathsf{ev\_one} & : & \mathsf{eval}\ (\mathsf{unit})\ \mathrm{D} \multimap \{\mathsf{return}\ (\mathsf{unit}')\ \mathrm{D}\}.
\end{array}
$$

Polymorphism, Sum Types, Void Type, Recursive Types. All of these can be added orthogonally and straightforwardly and are omitted here.

4.3 Suspensions with Memoization

The code below shows a technique for implementing lazy evaluation in the style of Haskell. This is slightly tricky, because the first time a suspension is accessed has be behave different from any future time it is accessed. In order to model this we make a linear assumption that, when consumed, makes an unrestricted assumption.

$$
\begin{array}{lcl}
\mathsf{susp} & : & \mathsf{tp} \to \mathsf{tp}. \\
\mathsf{delay} & : & \mathsf{exp}\ \mathrm{T} \to \mathsf{exp}\ (\mathsf{susp}\ \mathrm{T}). \\
\mathsf{force} & : & \mathsf{exp}\ (\mathsf{susp}\ \mathrm{T}) \to \mathsf{exp}\ \mathrm{T}. \\
\mathsf{thunk} & : & \mathsf{dest}\ \mathrm{T} \to \mathsf{val}\ (\mathsf{susp}\ \mathrm{T}). \\
\mathsf{read} & : & \mathsf{dest}\ \mathrm{T} \to \mathsf{dest}\ \mathrm{T} \to \mathsf{type}. \\
\mathsf{ev\_delay} & : & \mathsf{eval}\ (\mathsf{delay}\ \mathrm{E}_1)\ \mathrm{D} \\
 & & \quad \multimap \{\exists \mathsf{l}_1 {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{return}\ (\mathsf{thunk}\ \mathsf{l}_1)\ \mathrm{D} \\
 & & \qquad \otimes (\Pi \mathrm{D}' {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{read}\ \mathsf{l}_1\ \mathrm{D}' \\
 & & \qquad\quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
 & & \qquad\qquad \otimes (\Pi \mathrm{V}_1 {:} \mathsf{val}\ \mathrm{T}.\, \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \\
 & & \qquad\qquad\quad \multimap \{\mathsf{return}\ \mathrm{V}_1\ \mathrm{D}' \\
 & & \qquad\qquad\qquad \otimes\, !(\Pi \mathrm{D}'' {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{read}\ \mathsf{l}_1\ \mathrm{D}'' \multimap \{\mathsf{return}\ \mathrm{V}_1\ \mathrm{D}''\})\})\})\}. \\
\mathsf{ev\_force} & : & \mathsf{eval}\ (\mathsf{force}\ \mathrm{E}_1)\ \mathrm{D} \\
 & & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
 & & \qquad \otimes (\Pi \mathrm{L}_1 {:} \mathsf{dest}\ \mathrm{T}.\, \mathsf{return}\ (\mathsf{thunk}\ \mathrm{L}_1)\ \mathsf{d}_1 \multimap \{\mathsf{read}\ \mathrm{L}_1\ \mathrm{D}\})\}.
\end{array}
$$

4.4 State and Concurrency

So far, we have considered a wide range of pure, sequential programming language features. We were pleasantly surprised that our destination-passing style encoding may be extended to include mutable references and common concurrency primitives as well.

Futures. We now come to the first parallel construct: futures in the style of MultiLisp [Hal85], adapted to ML. There is no new type, since a future can be of any type. A destination D can now serve as a value (called promise D). If a promise is ever needed it needs to be available from then on, which is why we have a new family deliver V D which delivers value V to destination D and will be an unrestricted assumption.

Note that evaluating $\mathsf{future}\ E_1$ immediately returns a promise, while spawning a separate thread to compute E_1. This thread cannot communicate with other processes except through the final value it might deliver.

$$
\begin{array}{lll}
\mathsf{future} & : & \mathsf{exp}\ T \rightarrow \mathsf{exp}\ T. \\[4pt]
\mathsf{promise} & : & \mathsf{dest}\ T \rightarrow \mathsf{val}\ T. \\
\mathsf{deliver} & : & \mathsf{val}\ T \rightarrow \mathsf{dest}\ T \rightarrow \mathsf{type}. \\[4pt]
\mathsf{ev\_future} & : & \mathsf{eval}\ (\mathsf{future}\ E_1)\ D \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ T.\ \mathsf{return}\ (\mathsf{promise}\ \mathsf{d}_1)\ D \\
& & \qquad \otimes\ \mathsf{eval}\ E_1\ \mathsf{d}_1 \\
& & \qquad \otimes\ (\Pi V_1 : \mathsf{exp}\ T.\ \mathsf{return}\ V_1\ \mathsf{d}_1 \multimap \{!\mathsf{deliver}\ V_1\ \mathsf{d}_1\})\}. \\[4pt]
\mathsf{ret\_deliver} & : & \mathsf{return}\ V\ D \\
& & \quad \circ\!\!-\ \mathsf{return}\ (\mathsf{promise}\ D_1)\ D \\
& & \quad \leftarrow \mathsf{deliver}\ V\ D_1.
\end{array}
$$

In the last clause we use the reverse notation for implication in order to emphasize the operational reading in terms of backchaining. Previously, when we were trying to prove $\mathsf{return}\ V\ D$ for some given destination D, we would only succeed if it were directly present in the state, since there were no rules concluding $\mathsf{return}\ V\ D$, only the weaker $\{\mathsf{return}\ V\ D\}$.

With futures, we can also conclude $\mathsf{return}\ V\ D$ if there is a promise $\mathsf{promise}\ D_1$ with the right destination D that has delivered its answer (and therefore the unrestricted assumption $\mathsf{deliver}\ V\ D_1$ is available). Because futures can be iterated, delivering the results could require a chain of such inferences.

We find it remarkable how simple and modular the addition of futures to the prior semantic framework turned out to be.

Mutable References. Mutable references in the style of ML are easy to add: they just become linear hypotheses. The only novelty here is the higher-order encoding in terms of nested assumptions; otherwise the techniques and ideas are combined from LLF and Forum. New with respect to LLF is the destination-passing style and the representation of eval and return as assumptions rather than goals. New with respect to Forum is the monadic encapsulation and the presence of explicit proof terms.

$$
\begin{array}{lll}
\mathsf{ref} & : & \mathsf{tp} \rightarrow \mathsf{tp}. \\
\mathsf{newref} & : & \mathsf{exp}\ \mathrm{T} \rightarrow \mathsf{exp}\ (\mathsf{ref}\mathrm{T}). \\
\mathsf{assign} & : & \mathsf{exp}\ (\mathsf{ref}\ \mathrm{T}) \rightarrow \mathsf{exp}\ \mathrm{T} \rightarrow \mathsf{exp}\ (\mathsf{one}). \\
\mathsf{deref} & : & \mathsf{exp}\ (\mathsf{ref}\ \mathrm{T}) \rightarrow \mathsf{exp}\ \mathrm{T}.
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{cell} & : & \mathsf{dest}\ \mathrm{T} \rightarrow \mathsf{val}\ (\mathsf{ref}\ \mathrm{T}). \\
\mathsf{contains} & : & \mathsf{dest}\ \mathrm{T} \rightarrow \mathsf{val}\ \mathrm{T} \rightarrow \mathsf{type}.
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{ev\_newref} & : & \mathsf{eval}\ (\mathsf{newref}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ \mathrm{T}.\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \qquad \otimes (\Pi \mathrm{V}_1 : \mathsf{val}\ \mathrm{T}.\, \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \\
& & \qquad\quad \multimap \{\exists \mathsf{c} : \mathsf{dest}\ \mathrm{T}.\, \mathsf{contains}\ \mathsf{c}\ \mathrm{V}_1 \otimes \mathsf{return}\ (\mathsf{cell}\ \mathrm{C})\ \mathrm{D}\})\}. \\
\mathsf{ev\_assign} & : & \mathsf{eval}\ (\mathsf{assign}\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{ref}\ \mathrm{T}).\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \qquad \otimes (\Pi \mathrm{C}_1 : \mathsf{dest}\ \mathrm{T}.\, \mathsf{return}\ (\mathsf{cell}\ \mathrm{C}_1)\ \mathsf{d}_1 \\
& & \qquad\quad \multimap \{\exists \mathsf{d}_2 : \mathsf{dest}\ \mathrm{T}.\, \mathsf{eval}\ \mathrm{E}_2\ \mathsf{d}_2 \\
& & \qquad\qquad \otimes (\Pi \mathrm{V}_2 : \mathsf{val}\ \mathrm{T}.\, \mathsf{return}\ \mathrm{V}_2\ \mathsf{d}_2 \\
& & \qquad\qquad\quad \multimap \Pi \mathrm{V}_1 : \mathsf{val}\ \mathrm{T}.\, \mathsf{contains}\ \mathrm{C}_1\ \mathrm{V}_1 \\
& & \qquad\qquad\qquad \multimap \{\mathsf{contains}\ \mathrm{C}_1\ \mathrm{V}_2 \otimes \mathsf{return}\ (\mathsf{unit}')\ \mathrm{D}\})\})\}. \\
\mathsf{ev\_deref} & : & \mathsf{eval}\ (\mathsf{deref}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{ref}\ \mathrm{T}).\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \qquad \otimes (\Pi \mathrm{C}_1 : \mathsf{dest}\ \mathrm{T}.\, \mathsf{return}\ (\mathsf{cell}\ \mathrm{C}_1)\ \mathsf{d}_1 \\
& & \qquad\quad \multimap \Pi \mathrm{V}_1 : \mathsf{val}\ \mathrm{T}.\, \mathsf{contains}\ \mathrm{C}_1\ \mathrm{V}_1 \\
& & \qquad\qquad \multimap \{\mathsf{contains}\ \mathrm{C}_1\ \mathrm{V}_1 \otimes \mathsf{return}\ \mathrm{V}_1\ \mathrm{D}\})\}.
\end{array}
$$

If our language contained falsehood (0) and disjunction ($A \oplus B$), there would be an alternative formulation of dereferencing that does not consume and the re-create the linear assumption.

$$
\begin{array}{lll}
\mathsf{ev\_deref}' & : & \mathsf{eval}\ (\mathsf{deref}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{ref}\ \mathrm{T}).\, \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \qquad \otimes (\Pi \mathrm{C}_1 : \mathsf{dest}\ \mathrm{T}.\, \mathsf{return}\ (\mathsf{cell}\ \mathrm{C}_1)\ \mathsf{d}_1 \\
& & \qquad\quad \multimap \Pi \mathrm{V}_1 : \mathsf{val}\ \mathrm{T}.\, \{(\mathsf{contains}\ \mathrm{C}_1\ \mathrm{V}_1 \multimap \{0\}) \oplus \mathsf{return}\ \mathrm{V}_1\ \mathrm{D}\})\}.
\end{array}
$$

Concurrency. We now give an encoding of Concurrent ML as presented in [Rep99], omitting only negative acknowledgments. The representation can be extended to allow negative acknowledgments without changing its basic structure, but it would obscure the simplicity of representation.

We have two new type constructors, chan T for channels carrying values of type T, and event T for events of type T.

$$
\begin{array}{lll}
\mathsf{chan} & : & \mathsf{tp} \rightarrow \mathsf{tp}. \\
\mathsf{event} & : & \mathsf{tp} \rightarrow \mathsf{tp}.
\end{array}
$$

Processes are spawned with spawn E, where E is evaluated in the new process. spawn always returns the unit value. We synchronize on an event with sync. The primitive events are send and receive events for synchronous communication, as well as an event that is always enabled. In addition we have non-deterministic choice as in Milner's synchronous π-calculus, and wrappers that can be applied to event values. We avoid the usual ML style representation where unevaluated expressions of type T are presented as functions $\mathsf{arrow\ one}\ \mathrm{T}$. Instead, our semantics will simply not evaluate such expressions where appropriate. This leads to a more modular presentation of the language.

$$
\begin{array}{lcl}
\mathsf{spawn} & : & \mathsf{exp}\ \mathrm{T} \rightarrow \mathsf{exp}\ (\mathsf{one}). \\
\mathsf{sync} & : & \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}) \rightarrow \mathsf{exp}\ \mathrm{T}. \\
\\
\mathsf{channel} & : & \mathsf{exp}\ (\mathsf{chan}\ \mathrm{T}). \\
\\
\mathsf{alwaysEvt} & : & \mathsf{exp}\ \mathrm{T} \rightarrow \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{recvEvt} & : & \mathsf{exp}\ (\mathsf{chan}\ \mathrm{T}) \rightarrow \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{sendEvt} & : & \mathsf{exp}\ (\mathsf{chan}\ \mathrm{T}) \rightarrow \mathsf{exp}\ \mathrm{T} \rightarrow \mathsf{exp}\ (\mathsf{event}\ (\mathsf{one})). \\
\mathsf{choose} & : & \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}) \rightarrow \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}) \rightarrow \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{neverEvt} & : & \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{wrap} & : & \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}_1) \rightarrow (\mathsf{val}\ \mathrm{T}_1 \rightarrow \mathsf{exp}\ \mathrm{T}_2) \rightarrow \mathsf{exp}\ (\mathsf{event}\ \mathrm{T}_2).
\end{array}
$$

Among the new internals we find event values corresponding to the above events, plus channels that have been allocated with $\mathsf{channel}$.

$$
\begin{array}{lcl}
\mathsf{ch} & : & \mathsf{tp} \rightarrow \mathsf{type}. \\
\mathsf{chn} & : & \mathsf{ch}\ \mathrm{T} \rightarrow \mathsf{val}\ (\mathsf{chan}\ \mathrm{T}). \\
\\
\mathsf{alwaysEvt}' & : & \mathsf{val}\ \mathrm{T} \rightarrow \mathsf{val}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{recvEvt}' & : & \mathsf{val}\ (\mathsf{chan}\ \mathrm{T}) \rightarrow \mathsf{val}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{sendEvt}' & : & \mathsf{val}\ (\mathsf{chan}\ \mathrm{T}) \rightarrow \mathsf{val}\ \mathrm{T} \rightarrow \mathsf{val}\ (\mathsf{event}\ (\mathsf{one})). \\
\mathsf{choose}' & : & \mathsf{val}\ (\mathsf{event}\ \mathrm{T}) \rightarrow \mathsf{val}\ (\mathsf{event}\ \mathrm{T}) \rightarrow \mathsf{val}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{neverEvt}' & : & \mathsf{val}\ (\mathsf{event}\ \mathrm{T}). \\
\mathsf{wrap}' & : & \mathsf{val}\ (\mathsf{event}\ \mathrm{T}_1) \rightarrow (\mathsf{val}\ \mathrm{T}_1 \rightarrow \mathsf{exp}\ \mathrm{T}_2) \rightarrow \mathsf{val}\ (\mathsf{event}\ \mathrm{T}_2).
\end{array}
$$

Finally, we come to the operational semantics. There is a new type family, $\mathsf{synch}\ \mathrm{W}\ \mathrm{D}$ which synchronizes the event value W. The expression returned by the synchronization will eventually be evaluated and the resulting value returned to destination D.

We have chosen for processes that return to be explicitly "garbage collected" in the rule for spawn. Unlike for futures, the returned value is ignored.

$$
\begin{array}{lcl}
\mathsf{synch} & : & \mathsf{val}\ (\mathsf{event}\ \mathrm{T}) \rightarrow \mathsf{dest}\ \mathrm{T} \rightarrow \mathsf{type}. \\
\\
\mathsf{ev\_spawn} & : & \mathsf{eval}\ (\mathsf{spawn}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\mathsf{return}\ (\mathsf{unit}')\ \mathrm{D} \\
& & \qquad \otimes (\exists \mathsf{d}_1 {:} \mathsf{dest}.\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \qquad\quad \otimes (\Pi \mathrm{V}_1.\ \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \multimap \{1\}))\}. \\
\mathsf{ev\_sync} & : & \mathsf{eval}\ (\mathsf{sync}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 {:} \mathsf{dest}\ (\mathsf{event}\ \mathrm{T}).\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \qquad \otimes (\Pi \mathrm{W}_1 {:} \mathsf{val}\ (\mathsf{event}\ \mathrm{T}).\ \mathsf{return}\ \mathrm{W}_1\ \mathsf{d}_1 \\
& & \qquad\quad \multimap \{\mathsf{synch}\ \mathrm{W}_1\ \mathrm{D}\})\}. \\
\mathsf{ev\_channel} & : & \mathsf{eval}\ (\mathsf{channel})\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathrm{K} {:} \mathsf{ch}\mathrm{T}.\ \mathsf{return}\ (\mathsf{chn}\ \mathrm{K})\ \mathrm{D}\}.
\end{array}
$$

The rules for evaluating events are straightforward, since they simply evaluated the embedded expressions as needed and return the corresponding event value.

$$
\begin{array}{lll}
\mathsf{ev\_alwaysEvt} & : & \mathsf{eval}\ (\mathsf{alwaysEvt}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ \mathrm{T}.\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{V}_1 : \mathsf{val}\ \mathrm{T}.\ \mathsf{return}\ \mathrm{V}_1\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\mathsf{return}\ (\mathsf{alwaysEvt}'\ \mathrm{V}_1)\ \mathsf{d}_1\})\}. \\
\mathsf{ev\_recvEvt} & : & \mathsf{eval}\ (\mathsf{recvEvt}\ \mathrm{E}_1)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{chan}\ \mathrm{T}).\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{K} : \mathsf{ch}\ \mathrm{T}.\ \mathsf{return}\ (\mathsf{chn}\ \mathrm{K})\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\mathsf{return}\ (\mathsf{recvEvt}'\ (\mathsf{chn}\ \mathrm{K}))\ \mathrm{D}\})\}. \\
\mathsf{ev\_sendEvt} & : & \mathsf{eval}\ (\mathsf{sendEvt}\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{chan}\ \mathrm{T}).\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{K} : \mathsf{ch}\ \mathrm{T}.\ \mathsf{return}\ (\mathsf{chn}\ \mathrm{K})\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\exists \mathsf{d}_2 : \mathsf{dest}\ \mathrm{T}.\ \mathsf{eval}\ \mathrm{E}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad \otimes (\Pi \mathrm{V}_2 : \mathsf{val}\ \mathrm{T}.\ \mathsf{return}\ \mathrm{V}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad\quad \multimap \{\mathsf{return}\ (\mathsf{sendEvt}'\ (\mathsf{chn}\ \mathrm{K})\ \mathrm{V}_2)\ \mathrm{D}\})\})\}. \\
\mathsf{ev\_choose} & : & \mathsf{eval}\ (\mathsf{choose}\ \mathrm{E}_1\ \mathrm{E}_2)\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{event}\ \mathrm{T}).\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{W}_1 : \mathsf{val}\ (\mathsf{event}\ \mathrm{T}).\ \mathsf{return}\ \mathrm{W}_1\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\exists \mathsf{d}_2 : \mathsf{dest}\ (\mathsf{event}\ \mathrm{T}).\ \mathsf{eval}\ \mathrm{E}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad \otimes (\Pi \mathrm{W}_2 : \mathsf{val}\ (\mathsf{event}\ \mathrm{T}).\ \mathsf{return}\ \mathrm{W}_2\ \mathsf{d}_2 \\
& & \quad\quad\quad\quad\quad \multimap \{\mathsf{return}\ (\mathsf{choose}'\ \mathrm{W}_1\ \mathrm{W}_2)\ \mathrm{D}\})\})\}. \\
\mathsf{ev\_neverEvt} & : & \mathsf{eval}\ (\mathsf{neverEvt})\ \mathrm{D} \multimap \{\mathsf{return}\ (\mathsf{neverEvt}')\ \mathrm{D}\}. \\
\mathsf{ev\_wrap} & : & \mathsf{eval}\ (\mathsf{wrap}\ \mathrm{E}_1\ (\lambda \mathsf{x}.\ \mathrm{E}_2\ \mathsf{x}))\ \mathrm{D} \\
& & \quad \multimap \{\exists \mathsf{d}_1 : \mathsf{dest}\ (\mathsf{event}\ \mathrm{T}_1).\ \mathsf{eval}\ \mathrm{E}_1\ \mathsf{d}_1 \\
& & \quad\quad \otimes (\Pi \mathrm{W}_1 : \mathsf{val}\ (\mathsf{event}\ \mathrm{T}_1).\ \mathsf{return}\ \mathrm{W}_1\ \mathsf{d}_1 \\
& & \quad\quad\quad \multimap \{\mathsf{return}\ (\mathsf{wrap}'\ \mathrm{W}_1\ (\lambda \mathsf{x}.\ \mathrm{E}_2\ \mathrm{D}\})\}.
\end{array}
$$

Finally, we come to event matching as required for synchronization. We have a type family action which extracts a primitive event value (send, receive, or always) from a given complex event value (which may contain choices and wrappers). It also accumulates wrappers, returning a "continuation" $\mathsf{val}\ \mathrm{S} \to \mathsf{exp}\ \mathrm{T}$ for an original event of type T.

Note that action is don't-know non-deterministic in the manner of LLF or Lolli and is therefore written in the style of logic programming with reverse implications. Actions does not refer to the state.

$$
\mathsf{action} \ :\ \mathsf{val}\ (\mathsf{event}\ \mathrm{T}) \to \mathsf{val}\ (\mathsf{event}\ \mathrm{S}) \to (\mathsf{val}\ \mathrm{S} \to \mathsf{exp}\ \mathrm{T}) \to \mathsf{type}.
$$

$$
\begin{array}{lll}
\mathsf{act\_}\mathrm{T} & : & \mathsf{action}\ (\mathsf{alwaysEvt}'\ \mathrm{V})\ (\mathsf{alwaysEvt}'\ \mathrm{V})\ (\lambda \mathsf{x}.\ \mathsf{value}\ \mathsf{x}). \\
\mathsf{act\_!} & : & \mathsf{action}\ (\mathsf{sendEvt}'\ (\mathsf{chn}\ \mathrm{K})\ \mathrm{V})\ (\mathsf{sendEvt}'\ (\mathsf{chn}\ \mathrm{K})\ \mathrm{V})\ (\lambda \mathsf{x}.\ \mathsf{value}\ \mathsf{x}). \\
\mathsf{act\_?} & : & \mathsf{action}\ (\mathsf{rcvEvt}'\ (\mathsf{chn}\ \mathrm{K}))\ (\mathsf{rcvEvt}\ (\mathsf{chn}\ \mathrm{K}))\ (\lambda \mathsf{x}.\ \mathsf{value}\ \mathsf{x}). \\
\mathsf{act\_}\oplus_1 & : & \mathsf{action}\ (\mathsf{choose}'\ \mathrm{W}_1\ \mathrm{W}_2)\ \mathrm{A}\ (\lambda \mathsf{x}.\ \mathrm{E}\ \mathsf{x}) \\
& & \quad \leftarrow \mathsf{action}\ \mathrm{W}_1\ \mathrm{A}\ (\lambda \mathsf{x}.\ \mathrm{E}\ \mathsf{x}). \\
\mathsf{act\_}\oplus_2 & : & \mathsf{action}\ (\mathsf{choose}'\ \mathrm{W}_1\ \mathrm{W}_2)\ \mathrm{A}\ (\lambda \mathsf{x}.\ \mathrm{E}\ \mathsf{x}). \\
& & \quad \leftarrow \mathsf{action}\ \mathrm{W}_2\ \mathrm{A}\ (\lambda \mathsf{x}.\ \mathrm{E}\ \mathsf{x}). \\
\mathsf{act\_} \Rightarrow & : & \mathsf{action}\ (\mathsf{wrap}'\ \mathrm{W}\ (\lambda \mathsf{x}_2.\ \mathrm{E}_2\ \mathsf{x}_2))\ \mathrm{A}\ (\lambda \mathsf{x}_1.\ \mathsf{let}\ (\mathrm{E}_1\ \mathsf{x}_1)\ (\lambda \mathsf{x}_2.\ \mathrm{E}_2\ \mathsf{x}_2)) \\
& & \quad \leftarrow \mathsf{action}\ \mathrm{W}\ \mathrm{A}\ (\lambda \mathsf{x}_1.\ \mathrm{E}_1\ \mathsf{x}_1).
\end{array}
$$

The action rules are invoked when one or two synch assumptions are selected. Presumable the selection of such assumptions for concurrent evaluation is subject to fair scheduling. As further investigation of fairness and its consequences in this context is subject to further research and beyond the scope of this paper.

$$
\begin{array}{lcl}
\mathsf{synch}_1 & : & \mathsf{synch}\ \mathrm{W}\ \mathrm{D} \\
& & \quad \multimap \mathsf{action}\ \mathrm{W}\ (\mathsf{alwaysEvt}'\ \mathrm{V})\ (\lambda \mathsf{x}.\, \mathrm{E}\ \mathsf{x}) \\
& & \quad \rightarrow \{\mathsf{eval}\ (\mathrm{E}\ \mathrm{V})\ \mathrm{D}\}. \\
\mathsf{synch}_2 & : & \mathsf{synch}\ \mathrm{W}_1\ \mathrm{D}_1 \\
& & \quad \multimap \mathsf{synch}\ \mathrm{W}_2\ \mathrm{D}_2 \\
& & \quad \multimap \mathsf{action}\ \mathrm{W}_1\ (\mathsf{sendEvt}'\ (\mathsf{chn}\ \mathrm{K})\ \mathrm{V})\ (\lambda \mathsf{x}_1.\, \mathrm{E}_1\ \mathsf{x}_1) \\
& & \quad \rightarrow \mathsf{action}\ \mathrm{W}_2\ (\mathsf{rcvEvt}'\ (\mathsf{chn}\ \mathrm{K}))\ (\lambda \mathsf{x}_2.\, \mathrm{E}_2\ \mathsf{x}_2) \\
& & \quad \rightarrow \{\mathsf{eval}\ (\mathrm{E}_1\ \mathsf{unit}')\ \mathrm{D}_1 \otimes \mathsf{eval}\ (\mathrm{E}_2\ \mathrm{V})\ \mathrm{D}_2\}.
\end{array}
$$

Note that we use $\mathrm{A} \rightarrow \mathrm{B}$ instead of $\mathrm{A} \multimap \mathrm{B}$ whenever A does not require access to the current state. In this particular signature, $\mathrm{A} \multimap \mathrm{B}$ would in fact be equivalent, since no linear assumptions could be used in the proof of A (which is of the form $\mathsf{action}\ \mathrm{W}\ \mathrm{D}$ for some W and D.

5 Petri Nets in CLF

This sections applies CLF to encode Petri nets. We introduce preliminary multiset terminology in Section 5.1 and define two popular semantics, dubbed sequential and concurrent models, for Petri nets in Section 5.2. We give encodings for each of them in Sections 5.3 and 5.4, respectively. We conclude in Section 5.5 with some comments on how Petri nets can be encoded in LLF.

5.1 Multisets

Given a *support set* S, a *multiset* m over S is a collection of possibly repeated elements S. We formally define m as a function $m : S \rightarrow \mathbb{N}$ that associates a *multiplicity* to every element of S. We therefore write $\mathbb{N}^S$ for the set of all multisets over S.

Given a multiset m over S, any $a \in S$ such that $m(a) \geq 1$ is called an *element* of m. We denote this fact as $a \leftarrow m$. The *empty multiset*, written "$\cdot$" or "$\wr\,\wr$", has no elements: it is such that $\cdot(a) = 0$ for all $a \in S$. In analogy with the usual extensional notation for finite sets, where we write $\{a_1, \ldots, a_n\}$ for the set consisting of the elements $a_1, \ldots, a_n$, we will denote a multiset with elements $a_1, \ldots, a_n$ (possibly with replications) as $\wr a_1, \ldots, a_n \wr$. We extend this notation to the intensional construction of multisets given by the validity of a property: $\wr a : p(a)$ s.t. $a \leftarrow m \wr$ is the multiset consisting of all elements of m for which the property p holds.

The multiset-equivalent of the usual operations and relations on sets are inherited from the natural numbers $\mathbb{N}$. In particular, if m_1 and m_2 are two multisets with common support S,

- the *submultiset* relation, denoted $m_1 \leq m_2$, extends the "less than or equal to" relation over $\mathbb{N}$: $m_1 \leq m_2$ if for all $a \in S$, $m_1(a) \leq m_2(a)$.

- *multiset equality* is similarly lifted from the equality over $\mathbb{N}$. As for sets and numbers, $m_1 = m_2$ iff $m_1 \leq m_2$ and $m_2 \leq m_1$.

- the *multiset union* of m_1 and m_2 is the multiset $m = m_1 \uplus m_2$ such that for each $a \in S$, we have that $m(a) = m_1(a) + m_2(a)$.

- if $m_2 \leq m_1$, then the *multiset difference* of m_1 and m_2 is the multiset $m = m_1 - m_2$ such that, for all $a \in S$, $m(a) = m_1(a) - m_2(a)$.

- the *cardinality* of m_1, denoted $|m_1|$ is given as $|m_1| = \sum_{a \in S} m_1(a)$.

Other operations and relations are similarly defined, although we will not need them.

We will sometimes need to distinguish the different occurrences of an element a in a multiset m. For this purpose, we define a *labeled multiset* as a multiset m together with a set X and a function $\Lambda : X \to m$ that associates a unique *label* $x \in X$ to each occurrence of an element of m. We will always be in the position to choose the set X of labels in such a way that Λ is a multiplicity-conscious bijection between X and m, *i.e.,* $\wr \Lambda(x) : x \in X \wr = m$. We will take notational advantage of this flexibility and write $\boldsymbol{m}$ to indicate some labeled version of m. We will occasionally write $\boldsymbol{m}$ as $(X{:}m)$ to make explicit the set of labels X used in the construction of $\boldsymbol{m}$ from m. We will sometimes refer to a labeled multiset extensionally, writing for example $(x_1{:}a_1, \ldots, x_n{:}a_n)$ for a multiset $\wr a_1, \ldots, a_n \wr$ with labels $\{x_1, \ldots, x_n\}$. Labeled multisets inherit the operations and relations of their unlabeled cousins, although we shall be careful to preserve the multiplicity-conscious nature of the defining bijection (typically by renaming labels as necessary). Observe that, in the case of labeled multisets, these operations reduce to their set-theoretic counterparts.

Finally, we will occasionally view a multiset m as a sequence induced by some ordering (*e.g.,* alphabetic on the identifiers denoting the elements of its support set). We emphasize this reading by writing $\boldsymbol{m}$ as $\underline{m}$. We extend both concept and notation to labeled multisets in which case a secondary ordering over labels orders different occurrences of the same elements.

5.2 Petri Nets

A *Petri net* N is a directed bi-partite multigraph (P, T, E) whose two types of nodes, P and T, are called *places* and *transitions*, respectively [Pet62, Rei85, EW90]. The multi-edges E link places to transitions and transitions to place; each edge carries a multiplicity. It is convenient to take a transition-centric view of edges and define them as a function $E : T \to \mathbb{N}^P \times \mathbb{N}^P$ that associates each transition in $t \in T$ with a pair of multisets of places that we will denote $(\bullet t, t\bullet)$. The *pre-condition* $\bullet t$ of t lists the origin p of every edge that enters t; its multiplicity is expressed by the number of occurrences of the place p in $\bullet t$. The *post-condition* $t\bullet$ of t similarly represents the edges exiting t.

A *marking* $\boldsymbol{M}$ for N is a labeled multiset over P. Each element $x{:}p$ in $\boldsymbol{M}$ is called a *token*. Our use of labels constitutes a departure, known as the "individual token philosophy" and analyzed in [BMMS98, BM00], from the definition of markings and Petri nets commonly found in the literature. The implications are rather subtle: labels assign an identity to tokens inhabiting the same place, making it possible to distinguish them. Therefore, where the more traditional "collective token approach" prescribe removing "a" token from a place,

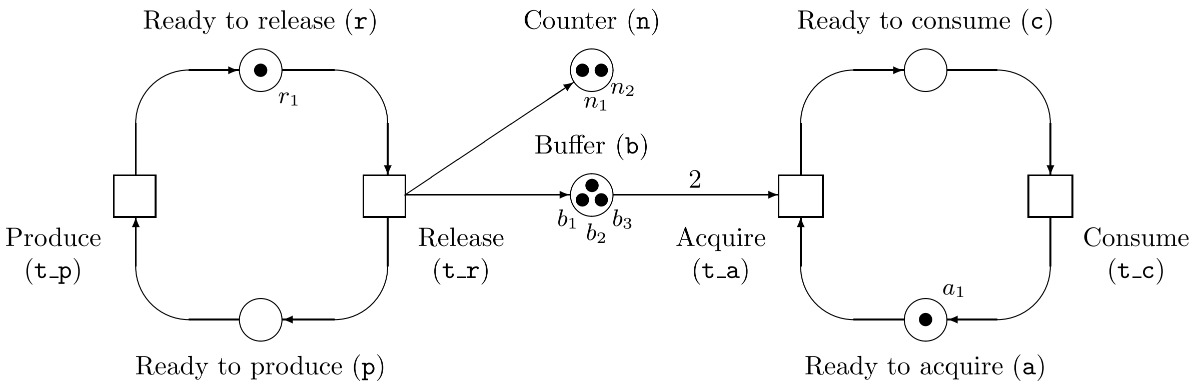

Figure 1: A Producer/Consumer Net

we shall say which of the possibly many token populating this place we are operating on, and each choice is accounted for as a different operation. Our more detailed formulation has the advantage of simplifying subsequent definitions (in particular traces) and will constitute the bases of our CLF encoding in Section 5.3. The drawback is that it distinguishes behaviors that the mainstream definition would identify.

Figure 1 describes a producer-consumer system N_{pc} by means of the traditional graphical representation for Petri nets: places are rendered as circles and transitions as squares; edge multiplicity is annotated on top of the edges unless it is 1 or 0 (in which case the edge is not drawn). The left cycle represents the producer, who releases one item per cycle, puts it in the common buffer and increments the counter by one. The consumer (on the right) extracts two items at a time from the buffer and consumes them. Figure 1 also specifies a marking $\boldsymbol{M}_{pc1}$ by putting a bullet for each occurrence of a place in the appropriate circle. Labels for these tokens cluster around the place where they occur.

A Petri net is a recipe for transforming a marking into another marking. More precisely, each transition can be seen as a description of how to modify a fragment of a marking. If two transitions do not try to rewrite the same portion of this marking, they can be applied in parallel. There are several ways to formalize this basic notion into a full blown semantics. We will examine two of them: first, the *interleaving semantics* expresses the global transformation as a sequence of application of transitions, recovering concurrency through permutations. Second, the *trace semantics* focuses on the dependencies (or lack thereof) among transitions.

5.2.1 Interleaving Semantics

Given a Petri net $N = (P, T, E)$ and a marking $\boldsymbol{M} = (X{:}M)$ over P, a transition $t \in T$ is *enabled* in $\boldsymbol{M}$ if $\bullet t \leq M$. If t is enabled at $\boldsymbol{M}$, then an *application* of t to $\boldsymbol{M}$ is defined as the judgment

$$\boldsymbol{M} \rhd_N \boldsymbol{M'}$$

where the marking $\boldsymbol{M'} = (X'{:}M')$ satisfies

$$\boldsymbol{M'} = ((X{:}M) - (\bullet x{:}\bullet t)) \uplus (x\bullet{:}t\bullet)$$

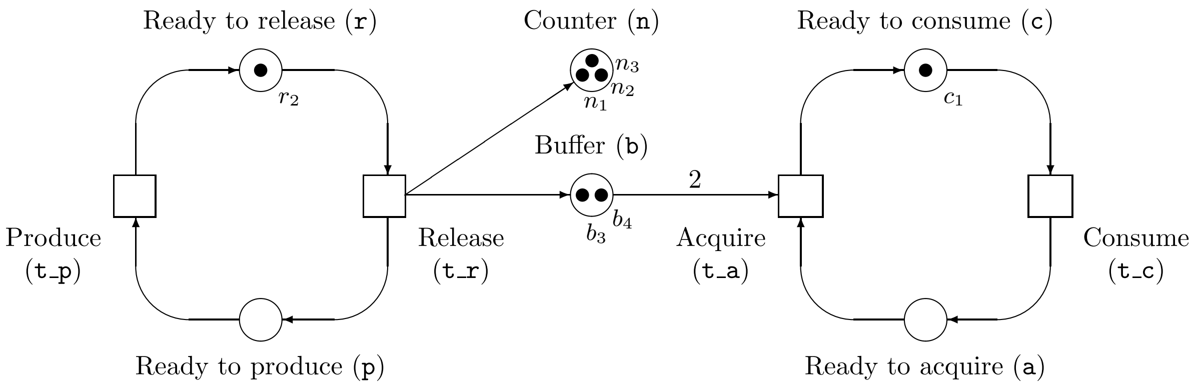

Figure 2: A Later Marking in the Producer/Consumer Net

where $\bullet x$ is the subset of the labels X associated with $\bullet t$ in $\boldsymbol{M}$ (note that it may not be unique), and $x\bullet$ is a set of new labels that are associated with $t\bullet$. The application of a transition to a marking is also called an *execution step* or the *firing* of t in $\boldsymbol{M}$.

Observe that, given N and $\boldsymbol{M}$, the triple $(t, \bullet x, x\bullet)$ identifies the application uniquely. We say that this triple *supports* the above application. We will generally annotate an execution step with the transition instance that supports it:

$$\boldsymbol{M} \rhd_N \boldsymbol{M'} \text{ by } (t, \bullet x, x\bullet).$$

An *execution* of a Petri net N from a marking $\boldsymbol{M_0}$ to a marking $\boldsymbol{M_n}$ is given by the reflexive and transitive closure of the execution step relation. It is denoted as

$$\boldsymbol{M_0} \rhd_N^* \boldsymbol{M_n} \text{ by } \mathcal{S}$$

where $\mathcal{S} = (t_1, \bullet x_1, x_1\bullet), \ldots, (t_n, \bullet x_n, x_n\bullet)$ and for $i = 1..n$, we have that $\boldsymbol{M_{i-1}} \rhd_N \boldsymbol{M_i}$ by $(t_i, \bullet x_i, x_i\bullet)$. Observe that the entire execution sequence, and in particular $\boldsymbol{M_n}$, is completely determined by $\boldsymbol{M_0}$ and the (possibly empty) sequence of transition applications $\mathcal{S}$ it is constructed from. Whenever $\boldsymbol{M_0} \rhd_N^* \boldsymbol{M_n}$ by $\mathcal{S}$ the marking $\boldsymbol{M_n}$ is said to be *reachable* from $\boldsymbol{M_0}$.

Figure 2 shows another marking $\boldsymbol{M}_{pc2}$ for the producer/consumer net. It is obtained by applying transitions `t_r`, `t_p` and `t_a` to the marking $\boldsymbol{M}_{pc1}$ embedded in Figure 1. This is described by the judgment:

$$\boldsymbol{M_1} \rhd_N^* \boldsymbol{M_2} \quad \text{by} \quad (\texttt{t\_r}, (r_1), (n_3, b_4, p_1)), \quad (\texttt{t\_p}, (p_1), (r_2)), \quad (\texttt{t\_a}, (a_1, b_1, b_2), (c_1))$$

Token p_1:`p` is produced by the first transition and consumed by the second. Observe that transition `t_a` consumes buffer tokens b_1 and b_2. We could have had it consume any combination of buffer tokens, including the token b_4 produced by the firing of `t_r` (the effect would have been to force a dependency between these two rules). This degree of discrimination does not arise in settings that keep tokens anonymous. Below, we will refer to this execution sequence as $\mathcal{S}_{pc}$.

Transition applications are completely ordered in a transition sequence. Still, some transitions are applied to independent portions of a given marking. We will now define concepts to highlight this form of concurrency, and prove some results for them.

A transition sequence $\mathcal{S} = (t_1, {\bullet}x_1, x_1{\bullet}), \ldots, (t_n, {\bullet}x_n, x_n{\bullet})$ is *well-labeled*, if for any label x appearing in $\mathcal{S}$:

- there is at most one index i such that $x \in x_i{\bullet}$;
- there is at most one index j such that $x \in {\bullet}x_j$;
- whenever there exist both i and j such that $x \in x_i{\bullet} \cap {\bullet}x_j$, then $i < j$.

It is clear that if $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$, then $\mathcal{S}$ is well-labeled.

If $\mathcal{S}_1$ and $\mathcal{S}_2$ are two well-labeled transition sequences, $\mathcal{S}_2$ is a *well-labeled exchange* of $\mathcal{S}_1$, written $\mathcal{S}_1 \sim_0 \mathcal{S}_2$, if $\mathcal{S}_1 = \mathcal{S}', (t, {\bullet}x, x{\bullet}), (t', {\bullet}x', x'{\bullet}), \mathcal{S}''$ and $\mathcal{S}_2 = \mathcal{S}', (t', {\bullet}x', x'{\bullet}), (t, {\bullet}x, x{\bullet}), \mathcal{S}''$. In our producer/consumer example, the execution sequence $\mathcal{S}'_{pc} = (\texttt{t\_r}, (r_1), (n_3, b_4, p_1))$, $(\texttt{t\_a}, (a_1, b_1, b_2), (c_1)), (\texttt{t\_p}, (p_1), (r_2))$ is a well-labeled exchange of $\mathcal{S}_{pc}$ above, but the sequence $\mathcal{S}^{\dagger}_{pc} = (\texttt{t\_p}, (p_1), (r_2)), (\texttt{t\_r}, (r_1), (n_3, b_4, p_1)), (\texttt{t\_a}, (a_1, b_1, b_2), (c_1))$ is not since p_1 occurs in the pre-condition of the first transition and in the post-condition of the second transition (violating the third condition).

It is clear that, if $\mathcal{S}_1 \sim_0 \mathcal{S}_2$, then

$$ {\bullet}x \cap x'{\bullet} \quad = \quad {\bullet}x' \cap x{\bullet} \quad = \quad {\bullet}x \cap {\bullet}x' \quad = \quad x{\bullet} \cap x'{\bullet} \quad = \quad \varnothing $$

since $\mathcal{S}_1$ and $\mathcal{S}_2$ are well-labeled. This condition is actually sufficient for two transitions to constitute a well-labeled exchange.

Lemma 5.1. (*Checking well-labeled exchange*)
Let $\mathcal{S}', (t, {\bullet}x, x{\bullet}), (t', {\bullet}x', x'{\bullet}), \mathcal{S}''$ be a well-labeled transition such that ${\bullet}x \cap x'{\bullet} = {\bullet}x' \cap x{\bullet} = {\bullet}x \cap {\bullet}x' = x{\bullet} \cap x'{\bullet} = \varnothing$, then $\mathcal{S}', (t, {\bullet}x, x{\bullet}), (t', {\bullet}x', x'{\bullet}), \mathcal{S}'' \sim_0 \mathcal{S}', (t', {\bullet}x', x'{\bullet}), (t, {\bullet}x, x{\bullet}), \mathcal{S}''$.

Proof. By induction on $\mathcal{S}'$ and then by verifying that the second sequence satisfies the definition of well-labeled transition. □

The well-labeled exchange relation is clearly symmetric. Moreover, if one of the sides supports an execution between two markings, the other relates these same two markings, as formalized in the following lemma.

Lemma 5.2. (*Well-labeled exchange preserves execution*)
If $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}_1$ and $\mathcal{S}_1 \sim_0 \mathcal{S}_2$, then $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}_2$.

Proof. Let $\mathcal{S}_1 = \mathcal{S}', (t, {\bullet}x, x{\bullet}), (t', {\bullet}x', x'{\bullet}), \mathcal{S}''$ and $\mathcal{S}_2 = \mathcal{S}', (t', {\bullet}x', x'{\bullet}), (t, {\bullet}x, x{\bullet}), \mathcal{S}''$.

By definition, there are markings $\bar{\boldsymbol{M}}$, $\boldsymbol{M}^*$ and $\bar{\boldsymbol{M}}'$ such that

$$ \boldsymbol{M} \rhd_N^* \bar{\boldsymbol{M}} \text{ by } \mathcal{S}', \quad \bar{\boldsymbol{M}} \rhd_N \boldsymbol{M}^* \text{ by } (t, {\bullet}x, x{\bullet}), \quad \boldsymbol{M}^* \rhd_N \bar{\boldsymbol{M}}' \text{ by } (t', {\bullet}x', x'{\bullet}), \quad \bar{\boldsymbol{M}}' \rhd_N^* \boldsymbol{M'} \text{ by } \mathcal{S}'' $$

where $\boldsymbol{M}^* = (\bar{\boldsymbol{M}} - ({\bullet}x{:}{\bullet}t)) \uplus (x{\bullet}{:}t{\bullet})$ and $\bar{\boldsymbol{M}}' = (\boldsymbol{M}^* - ({\bullet}x'{:}{\bullet}t')) \uplus (x'{\bullet}{:}t'{\bullet}) = (((\bar{\boldsymbol{M}} - ({\bullet}x{:}{\bullet}t)) \uplus (x{\bullet}{:}t{\bullet})) - ({\bullet}x'{:}{\bullet}t')) \uplus (x'{\bullet}{:}t'{\bullet})$.

Since $\mathcal{S}_1$ and $\mathcal{S}_2$ are well-labeled, $x{\bullet} \cap {\bullet}x' = \varnothing$, therefore, we can rewrite $\bar{\boldsymbol{M}}'$ as $\bar{\boldsymbol{M}}' = (\bar{\boldsymbol{M}} - ({\bullet}x{:}{\bullet}t) - ({\bullet}x'{:}{\bullet}t')) \uplus (x{\bullet}{:}t{\bullet}) \uplus (x'{\bullet}{:}t'{\bullet})$, or as

$$ \bar{\boldsymbol{M}}' = ((\underbrace{(\bar{\boldsymbol{M}} - ({\bullet}x'{:}{\bullet}t')) \uplus (x'{\bullet}{:}t'{\bullet})}_{\boldsymbol{M}^{**}}) - ({\bullet}x{:}{\bullet}t)) \uplus (x{\bullet}{:}t{\bullet}). $$

Let then $\boldsymbol{M^{**}} = (\bar{\boldsymbol{M}} - (\bullet x'{:}\bullet t')) \uplus (x'\bullet{:}t'\bullet)$. We have then that $\bar{\boldsymbol{M}} \rhd_N \boldsymbol{M^{**}}$ by $(t, \bullet x', x'\bullet)$ and $\boldsymbol{M^{**}} \rhd_N \bar{\boldsymbol{M}}'$ by $(t', \bullet x, x\bullet)$, from which we deduce that $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}_2$. □

Two well-labeled transition sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ are *equivalent*, written $\mathcal{S}_1 \sim \mathcal{S}_2$, if $\mathcal{S}_1$ is a permutation of $\mathcal{S}_2$. We will sometimes write $\mathcal{S}_2 = \pi(\mathcal{S}_1)$ where π is the witnessing permutation. It is easy to show that $\sim$ is indeed an equivalence relation. In our recurring example, the transition sequence $\mathcal{S}''_{pc} = (\texttt{t\_a}, (a_1, b_1, b_2), (c_1)), (\texttt{t\_r}, (r_1), (n_3, b_4, p_1)), (\texttt{t\_p}, (p_1), (r_2))$ is a equivalent to $\mathcal{S}_{pc}$.

Permutations can be expressed as iterated exchanges of two adjacent elements. Sorting algorithms such as bubble-sort are based on this property. It is therefore not surprising that showing the equivalence of two transition sequences can be reduced to producing a sequence of well-labeled exchanges.

Lemma 5.3. (*Well-labeled equivalence maps to iterated well-labeled exchange*)
If $\mathcal{S} \sim \mathcal{S}'$, then there are transition sequences $\mathcal{S}_0, \ldots, \mathcal{S}_n$ such that $\mathcal{S} = \mathcal{S}_0$, $\mathcal{S}' = \mathcal{S}_n$, and for $i = 1..n$, $\mathcal{S}_{i-1} \sim_0 \mathcal{S}_i$.

Proof. Let $\mathcal{S} = \xi_1, \ldots, \xi_k$, with $\xi_j = (t_j, \bullet x_j, x_j\bullet)$ for $j = 1..k$. Then, by definition, $\mathcal{S}' = \xi_{\pi(1)}, \ldots, \xi_{\pi(k)}$ for some permutation π. We define the distance between $\mathcal{S}$ and $\mathcal{S}'$ as the tuple $d(\mathcal{S}, \mathcal{S}') = (d_1(\mathcal{S}, \mathcal{S}'), \ldots, d_k(\mathcal{S}, \mathcal{S}'))$ where $d_j(\mathcal{S}, \mathcal{S}') = |j - \pi(j)|$, for $j = 1..k$. We then proceed by lexicographic induction on $d(\mathcal{S}, \mathcal{S}')$.

If $d(\mathcal{S}, \mathcal{S}') = \vec{0}$, then $\mathcal{S} = \mathcal{S}'$ and $n = 0$.

Otherwise, let l be the first non-zero index in $d(\mathcal{S}, \mathcal{S}')$. Then,

$$\mathcal{S} = \bar{\mathcal{S}}, \xi_l, \tilde{\mathcal{S}} \qquad \text{and} \qquad \mathcal{S}' = \bar{\mathcal{S}}, \mathcal{S}', \xi'_{\pi(l)-1}, \xi'_{\pi(l)}, \mathcal{S}^{**}$$

Let $\mathcal{S}'' = \bar{\mathcal{S}}, \mathcal{S}', \xi'_{\pi(l)}, \xi'_{\pi(l)-1}, \mathcal{S}^{**}$. Since $\mathcal{S}$ and $\mathcal{S}'$ are well-labeled and equivalent, $\xi'_{\pi(l)-1}$ and $\xi'_{\pi(l)}$ must be independent. Thus, we have that $\mathcal{S}'' \sim_0 \mathcal{S}'$.

Notice that $d(\mathcal{S}, \mathcal{S}'') < d(\mathcal{S}, \mathcal{S}')$ since $d_l(\mathcal{S}, \mathcal{S}'') = d_l(\mathcal{S}, \mathcal{S}') - 1$ (and $d_j(\mathcal{S}, \mathcal{S}'')$ is still null for $j < l$). Then, by induction hypothesis, there are transition sequences $\mathcal{S}_0, \ldots, \mathcal{S}_n$ such that $\mathcal{S} = \mathcal{S}_0$, $\mathcal{S}'' = \mathcal{S}_n$, and for $i = 1..n$, $\mathcal{S}_{i-1} \sim_0 \mathcal{S}_i$. Then, simply add $\mathcal{S}'' \sim_0 \mathcal{S}'$ to this sequence to obtain the desired result □

In our producer/consumer example, we observed that $\mathcal{S}_{pc} \sim \mathcal{S}''_{pc}$. This can be unfolded as $\mathcal{S}_{pc} \sim_0 \mathcal{S}'_{pc} \sim_0 \mathcal{S}''_{pc}$.

This result, together with Lemma 5.4, allows for a simple proof of the fact that $\sim$ preserves marking reachability.

Lemma 5.4. (*Well-labeled equivalence preserves execution*)
If $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$ and $\mathcal{S} \sim \mathcal{S}'$, then $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}'$.

Proof. By lemma 5.3, there are transition sequences $\mathcal{S}_0, \ldots, \mathcal{S}_n$ such that $\mathcal{S} = \mathcal{S}_0$, $\mathcal{S}' = \mathcal{S}_n$, and for $i = 1..n$, $\mathcal{S}_{i-1} \sim_0 \mathcal{S}_i$. We now proceed by induction on n.

If $n = 0$, then $\mathcal{S} = \mathcal{S}'$ and the desired result is trivially satisfied.

For $n \geq 1$, we have a sequence of the form $\mathcal{S}_0, \ldots, \mathcal{S}_{n-1}, \mathcal{S}_n$. By induction hypothesis, there is an execution $\boldsymbol{M} \rhd_N^* \boldsymbol{M''}$ by $\mathcal{S}_{n-1}$ for some marking $\boldsymbol{M''}$. By Lemma 5.2, $\boldsymbol{M''} \rhd_N \boldsymbol{M'}$ by $\mathcal{S}_n$. From this the desired result follows easily. □

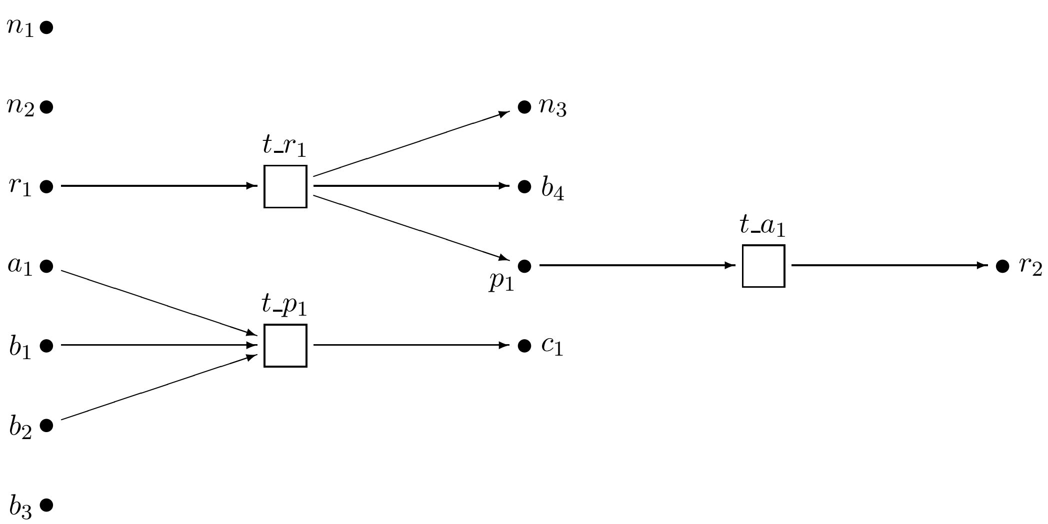

Figure 3: A Trace in the Producer/Consumer Net

5.2.2 Trace Semantics

Given a Petri net $N = (P, T, E)$, a *trace* $\mathcal{T}$ of N is an acyclic directed bi-partite graph $(\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E})$ such that $\bar{\boldsymbol{P}} = (X:\bar{P})$ is a labeled multiset over P, $\bar{\boldsymbol{T}} = (Y:\bar{T})$ is a labeled multiset over T, and $\mathbf{E}$ is a set of (single) edges $\mathbf{E} : \bar{\boldsymbol{T}} \to \mathbf{2}^{\bar{\boldsymbol{P}}} \times \mathbf{2}^{\bar{\boldsymbol{P}}}$. In this report, we will assume that $\bar{\boldsymbol{P}}$, $\bar{\boldsymbol{T}}$ and $\mathbf{E}$ are finite. Similarly to the case of Petri nets, we will denote the two components of the image of a labeled transition $(y:t)$ according to $\mathbf{E}$ as $\bullet(y:t)$ and $(y:t)\bullet$, respectively. $\mathbf{E}$ is subject to the following restrictions:

1. For any transition $t \in T$ such that $(y:t) \in \bar{\boldsymbol{T}}$, if $\bullet(y:t) = (X_1:\bar{P}_1)$ and $(y:t)\bullet = (X_2:\bar{P}_2)$ (with $\bar{P}_1, \bar{P}_2 \leq \bar{P}$ and X_1 and X_2 disjoint sets of labels), then $\bar{P}_1 = \bullet t$ and $\bar{P}_2 = t\bullet$.

2. For every $(x:p) \in \bar{\boldsymbol{P}}$ and $(y_1:t_1), (y_2:t_2) \in \bar{\boldsymbol{T}}$,

   (a) if $(x:p) \in \bullet(y_1:t_1)$ and $(x:p) \in \bullet(y_2:t_2)$, then $(y_1:t_1) = (y_2:t_2)$.

   (b) if $(x:p) \in (y_1:t_1)\bullet$ and $(x:p) \in (y_2:t_2)\bullet$, then $(y_1:t_1) = (y_2:t_2)$.

The first restriction forces each occurrence of a transition t in the trace $\mathcal{T}$ to be consistent with its definition in N: its incoming edges should originate from places in $\bar{\boldsymbol{P}}$ of the same type and in the same number as specified in $\bullet t$, and similarly for outgoing edges. The second restriction requires a place in $\mathcal{T}$ to be produced at most once and to be consumed at most once.

Given a trace $\mathcal{T}$, the *initial marking* of $\mathcal{T}$, denoted $\bullet\mathcal{T}$, is the set of labeled places in $\bar{\boldsymbol{P}}$ without any incoming edge: $\bullet\mathcal{T} = \{(x:p) : \text{there is no } (y:t) \in \bar{\boldsymbol{T}} \text{ s.t. } (x,t) \in \bullet(y:t)\}$. The *final marking* of $\mathcal{T}$ is similarly defined as the set of labeled places in $\mathcal{T}$ without any outgoing edge. It is denoted $\mathcal{T}\bullet$.

Figure 3 shows a trace $\mathcal{T}_{pc}$ for the consumer/producer example in this section. For clarity, we have omitted the identifiers for places and transition, relying only on mnemonic label names. The initial marking of $\mathcal{T}$ consists of the places $\bullet\mathcal{T}_{pc} = \{n_1, n_2, r_1, a_1, c_1, b_1, b_2, b_3\}$

while its final marking is $\mathcal{T}_{pc}\bullet = \{n_1, n_2, n_3, b_4, r_2, a_1, c_2, b_3\}$. These two sets of tokens correspond to the markings displayed in Figures 1 and 2, respectively. This trace consists of three transition instances: $(t\_r_1{:}\texttt{t\_r})$, $(t\_a_1{:}\texttt{t\_a})$ and $(t\_p_1{:}\texttt{t\_p})$.

5.2.3 Equivalence

We now show that the two presentations of the semantics of a Petri net are equivalent from the point of view of reachability: whenever there is an execution sequence between two markings, they are the initial and final marking of some trace, and for every trace there is an execution sequence between its initial and final marking. Besides of being of interesting by itself, this fact will help us encode the notion of traces in CLF. The above intuition is formalized in the following property.

Property 5.5. (*Equivalence of interleaving and trace semantics*)
Let $N = (P, T, E)$ be a Petri net.

1. *Given markings $\boldsymbol{M}$ and $\boldsymbol{M'}$ over N such that $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$* by $\mathcal{S}$*, then there is a trace $\mathcal{T}$ over N such that $\bullet\mathcal{T} = \boldsymbol{M}$ and $\mathcal{T}\bullet = \boldsymbol{M'}$.*
2. *Given a trace $\mathcal{T}$ over N, then there is a valid execution sequence $\mathcal{S}$ such that $\bullet\mathcal{T} \rhd_N^* \mathcal{T}\bullet$* by $\mathcal{S}$*.*

Proof.

1. The proof proceeds by induction on the sequence $\mathcal{S} = (t_1, \bullet x_1, x_1\bullet), \ldots, (t_n, \bullet x_n, x_n\bullet)$ of execution steps that define the construction of $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$.

   If $\mathcal{S}$ is empty, then $\boldsymbol{M'} = \boldsymbol{M}$. Then the triple $\mathcal{T} = (\boldsymbol{M}, \varnothing, \varnothing)$ satisfies the definition of a trace, and moreover $\bullet\mathcal{T} = \mathcal{T}\bullet = \boldsymbol{M}$.

   Assume now that $\mathcal{S} = \mathcal{S}^*, (t, \bullet x, x\bullet)$. Then, there is a marking $\boldsymbol{M^*}$ such that $\boldsymbol{M} \rhd_N^* \boldsymbol{M^*}$ by $\mathcal{S}^*$, and $\boldsymbol{M^*} \rhd_N \boldsymbol{M'}$ by $(t, \bullet x, x\bullet)$. By induction hypothesis there is a trace $\mathcal{T}^* = (\bar{\boldsymbol{P}}^*, \bar{\boldsymbol{T}}^*, \mathbf{E}^*)$ such that $\bullet\mathcal{T}^* = \boldsymbol{M}$ and $\mathcal{T}^*\bullet = \boldsymbol{M^*}$. By definition of execution step, $\boldsymbol{M'} = (\boldsymbol{M^*} - (\bullet x{:}\bullet t)) \uplus (x\bullet{:}t\bullet)$. We then define $\mathcal{T}' = (\bar{\boldsymbol{P}}', \bar{\boldsymbol{T}}', \mathbf{E}')$ as follows:

   - $\bar{\boldsymbol{P}}' = \bar{\boldsymbol{P}}^* \cup (x\bullet{:}t\bullet)$.
   - $\bar{\boldsymbol{T}}' = \bar{\boldsymbol{T}}^* \cup \{(y{:}t)\}$, for some new label y.
   - $\mathbf{E}' = \mathbf{E}^* \cup \{(y{:}t) \mapsto ((\bullet x{:}\bullet t), (x\bullet{:}t\bullet))\}$.

   Observe that $\bullet(y{:}t) = (\bullet x{:}\bullet t)$ and $(y{:}t)\bullet = (x\bullet{:}t\bullet)$.

   We shall prove that $\mathcal{T}'$ is indeed a trace. Condition (1) follows by construction. Condition (2-a) is satisfied since $\bullet(y{:}t) \subseteq \mathcal{T}^*\bullet$. Finally, condition (2-b) holds since the labels in $x\bullet$ are new.

   In order to conclude this case of the proof, we observe that $\bullet\mathcal{T}' = \boldsymbol{M}$ since the construction of $\mathcal{T}'$ from $\mathcal{T}^*$ did not involve the introduction of any incoming edge to a place, and that $\mathcal{T}'\bullet = \boldsymbol{M'}$ because the set of places without an outgoing edge has been upgraded to exclude $(\bullet x{:}\bullet t)$ and extended with $(x\bullet t\bullet)$, which is exactly how $\boldsymbol{M'}$ has been produced

2. Since traces are acyclic, their nodes and edges naturally define a partial order upon which to base a proof by induction. Therefore, we will proceed by induction on the structure of $\mathcal{T}$.

   Our base case captures traces without any transition nodes. Therefore $\mathcal{T} = (\bar{\boldsymbol{P}}, \varnothing, \varnothing)$, for some labeled multiset of places $\bar{\boldsymbol{P}}$. Then $\bullet\mathcal{T} = \mathcal{T}\bullet = \bar{\boldsymbol{P}}$. Our statement is proved by taking $\mathcal{S}$ to be the empty sequence of transitions.

   Assume now that $\mathcal{T} = (\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E})$ contains at least one transition node $(\hat{y}{:}\hat{t})$. Then, there is at least one node $(y{:}t)$ such that $(y{:}t)\bullet \subseteq \mathcal{T}\bullet$. In order to show this, construct the sequence σ of transition nodes as follows: initialize σ to $(\hat{y}{:}\hat{t})$. Let $(y'{:}t')$ be the last element in σ.

   (a) If $(y'{:}t')\bullet \subseteq \mathcal{T}\bullet$, then $(y'{:}t')$ is the desired transition $(y{:}t)$.

   (b) Otherwise, there is a labeled place $(x{:}p)$ and a labeled transition $(y''{:}t'')$ such that $(y'{:}t')\bullet \ni (x{:}p) \in \bullet(y''{:}t'')$. In this case, extend σ with $(y''{:}t'')$ and repeat.

   Since $\mathcal{T}$ is acyclic (and finite), this procedure can make use of (b) only a finite number of times before falling back on (a).

   Let therefore $(y{:}t) \in \bar{\boldsymbol{T}}$ be one of the transition nodes in $\mathcal{T}$ such that $(y{:}t)\bullet \subseteq \mathcal{T}\bullet$. We define the trace $\mathcal{T}' = (\bar{\boldsymbol{P}}', \bar{\boldsymbol{T}}', \mathbf{E}')$ as follows:

   - $\bar{\boldsymbol{P}}' = \bar{\boldsymbol{P}} \setminus (y{:}t)\bullet$.
   - $\bar{\boldsymbol{T}}' = \bar{\boldsymbol{T}} \setminus \{(y{:}t)\}$.
   - $\mathbf{E}' = \mathbf{E} \setminus \{(y{:}t) \mapsto (\bullet(y{:}t), (y{:}t)\bullet)\}$.

   It is a simple exercise to verify that $\mathcal{T}'$ is a trace. Moreover, $\bullet\mathcal{T} = \bullet\mathcal{T}'$ and $\mathcal{T}\bullet = (\mathcal{T}\bullet' \setminus \bullet(y{:}t)) \cup (y{:}t)\bullet$.

   By induction hypothesis, there is a transition sequence $\mathcal{S}'$ such that $\bullet\mathcal{T}' \rhd_N^* \mathcal{T}\bullet'$ by $\mathcal{S}'$. Let then $\bullet(y{:}t) = (\bullet x{:}\bullet t)$ and $(y{:}t)\bullet = (x\bullet{:}t\bullet)$. Then, the desired transition sequence $\mathcal{S}$ is defined as $\mathcal{S}', (t, \bullet x, x\bullet)$.

This concludes the sketch of this proof. □

This proof outlines a method for building a trace out of an execution sequence. It is worth making it more explicit. The trace associated with a transition sequence $\mathcal{S}$ and a marking $\boldsymbol{M}$, denoted $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$, is defined as follows:

$$
\begin{array}{lll}
\mathcal{T}_{\boldsymbol{M}}(\cdot) & = & (\boldsymbol{M}, \varnothing, \varnothing) \\
\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}, (t, \bullet x, x\bullet)) & = & (\bar{\boldsymbol{P}} \cup (x\bullet{:}t\bullet), \qquad\qquad \text{where } \mathcal{T}_{\boldsymbol{M}}(\mathcal{S}) = (\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E}) \\
 & & \quad \bar{\boldsymbol{T}} \cup \{(y{:}t)\}, \qquad\qquad \text{and } y \text{ is a new label} \\
 & & \quad \mathbf{E} \cup \{(y{:}t) \mapsto ((\bullet x{:}\bullet t), (x\bullet{:}t\bullet))\})
\end{array}
$$

Applying this definition to the producer/consumer net, it is easy to check that $\mathcal{T}_{\boldsymbol{M}_{pc1}}(\mathcal{S}_{pc})$ is a trace that differ from $\mathcal{T}_{pc}$ only by the the name of labels not occurring in $\boldsymbol{M}_{pc1}$.

Indirectly, this definition provides a constructive method for extending a trace $\mathcal{T}$ with respect to a Petri net N: identify a transition t in N such that $\mathcal{T}\bullet = (\bullet x{:}\bullet t), \bar{\boldsymbol{P}}'$; add the transition node $(y{:}t)$ (for some new label y) and place nodes $(x\bullet{:}t\bullet)$ for some new labels $x\bullet$; add edges from $(\bullet x{:}\bullet t)$ to $(y{:}t)$ and from $(y{:}t)$ to $(x\bullet{:}t\bullet)$.

The above proof also outlines a method for flattening a trace $\mathcal{T}$ into an execution sequence $\mathcal{S}$: repeatedly pull out a labeled transition with only initial places in its precondition until all transitions in $\mathcal{T}$ have been processed in this way. The following definition generalizes this technique by describing the set of transition sequences $\mathcal{S}(\mathcal{T})$ that can be extracted in this way from a trace $\mathcal{T}$.

$$
\begin{array}{lll}
(\cdot) \in \mathcal{S}((\bar{\boldsymbol{P}}, \varnothing, \varnothing)) & & \\
\mathcal{S}, (t, \bullet x, x\bullet) \in \mathcal{S}(\mathcal{T}) & \text{if} & \mathcal{T} = (\bar{\boldsymbol{P}} \cup (y{:}t)\bullet, \bar{\boldsymbol{T}} \cup \{(y{:}t)\}, \mathbf{E} \cup \{(y{:}t) \mapsto (\bullet(y{:}t), (y{:}t)\bullet)\}) \\
& \text{and} & (y{:}t)\bullet \subseteq \mathcal{T}\bullet \\
& \text{and} & \bullet(y{:}t) = (\bullet x{:}\bullet t) \\
& \text{and} & (y{:}t)\bullet = (x\bullet{:}t\bullet) \\
& \text{and} & \mathcal{S} \in \mathcal{S}((\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E}))
\end{array}
$$

$\mathcal{S} \in \mathcal{S}(\mathcal{T})$ is called a transition sequence *underlying* the trace $\mathcal{T}$. In the producer/consumer example, it is easy to check that $\mathcal{S}(\mathcal{T}_{pc}) = \{\mathcal{S}_{pc}, \mathcal{S}'_{pc}, \mathcal{S}''_{pc}\}$.

The Equivalence Property 5.5 can then be specialized to these constructions. The proofs of the following two corollaries can be excised from the proof of this property.

Corollary 5.6. (*From transition sequences to traces*)
If $\boldsymbol{M} \rhd^*_N \boldsymbol{M'}$ by $\mathcal{S}$, *then* $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$ *is a trace. Moreover,* $\boldsymbol{M} = \bullet(\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}))$ *and* $\boldsymbol{M'} = (\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}))\bullet$.

Corollary 5.7. (*From traces to transition sequences*)
If $\mathcal{T}$ *is a trace, then for every* $\mathcal{S} \in \mathcal{S}(\mathcal{T})$ *and* $\bullet\mathcal{T} \rhd^*_N \mathcal{T}\bullet$ by $\mathcal{S}$.

It is easy to shows that the two above constructions are essentially inverse of each other.

Lemma 5.8. ($\mathcal{S}(\_)$ *and* $\mathcal{T}_\_(\_)$ *are each other's inverse*)
Let N *be a Petri net.*

1. *If* $\mathcal{T}$ *is a trace and* $\mathcal{S} \in \mathcal{S}(\mathcal{T})$, *then* $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$ *is isomorphic to* $\mathcal{T}$ *for* $\boldsymbol{M} = \bullet\mathcal{T}$.
2. *If* $\boldsymbol{M} \rhd^*_N \boldsymbol{M'}$ by $\mathcal{S}$, *then* $\mathcal{S} \in \mathcal{S}(\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}))$.

Proof.

1. This part of the proof proceeds by induction on the structure of $\mathcal{S}$. Assume first that $\mathcal{S} = \cdot$, then $\mathcal{T} = (\bar{\boldsymbol{P}}, \varnothing, \varnothing)$ and $\bullet\mathcal{T} = \bar{\boldsymbol{P}}$. By definition, we have that $\mathcal{T}_{\bar{\boldsymbol{P}}}(\cdot) = (\bar{\boldsymbol{P}}, \varnothing, \varnothing)$, which is clearly isomorphic to $\mathcal{T}$.

   Assume now that $\mathcal{S} = \mathcal{S}', (t, \bullet x, x\bullet)$. Then, it must be the case that

   $$\mathcal{T} = (\bar{\boldsymbol{P}} \cup (y{:}t)\bullet, \bar{\boldsymbol{T}} \cup \{(y{:}t)\}, \mathbf{E} \cup \{(y{:}t) \mapsto (\bullet(y{:}t), (y{:}t)\bullet)\})$$

   with $(y{:}t)\bullet \subseteq \mathcal{T}\bullet$, $\bullet(y{:}t) = (\bullet x{:}\bullet t)$, $(y{:}t)\bullet = (x\bullet{:}t\bullet)$ and $\mathcal{S} \in \mathcal{S}(\mathcal{T}')$ where $\mathcal{T}' = (\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E})$.

   By induction hypothesis, $\mathcal{T}_{\boldsymbol{M'}}(\mathcal{S}')$ is isomorphic to $\mathcal{T}'$ where $\boldsymbol{M'} = \bullet\mathcal{T}'$. Let then $\boldsymbol{M} = \bullet\mathcal{T}$; we have that $\boldsymbol{M'} = \boldsymbol{M} - (\bullet(y{:}t) \cap \boldsymbol{M})$. Let $\mathcal{T}_{\boldsymbol{M'}}(\mathcal{S}') = (\bar{\boldsymbol{P}}', \bar{\boldsymbol{T}}', \mathbf{E}')$.

By construction, we have that

$\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}) = \mathcal{T}_{\boldsymbol{M}}(\mathcal{S}', (t, \bullet x, x\bullet)) = (\bar{\boldsymbol{P}}' \cup (x\bullet{:}t\bullet), \bar{\boldsymbol{T}}' \cup \{(y'{:}t)\}, \mathbf{E}' \cup \{(y'{:}t) \mapsto ((\bullet x{:}\bullet t), (x\bullet{:}t\bullet))\})$.

We can then extend the isomorphism between $\mathcal{T}_{\boldsymbol{M'}}(\mathcal{S}')$ and $\mathcal{T}'$ to $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$ and $\mathcal{T}$ by relating the added places with identical labels, relating the added transitions labeled y and y' respectively, and relating the added edges in $\mathbf{E}$ and $\mathbf{E}'$ respectively.

2. The proof proceeds by induction on the structure of $\mathcal{S}$. The case in which $\mathcal{S} = \cdot$ is trivial. The inductive case, in which $\mathcal{S} = \mathcal{S}', (t, \bullet x, x\bullet)$ is handled similarly to the proof of the first part of this lemma: we unfold $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$, deduce by induction hypothesis that $\mathcal{S}' \in \mathcal{S}(\mathcal{T}_{\boldsymbol{M'}}(\mathcal{S}'))$ for $\boldsymbol{M'} = \boldsymbol{M} - (\bullet(y{:}t) \cap \boldsymbol{M})$, and then show that $\mathcal{S} \in \mathcal{S}(\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}))$.

This concludes the proof of this lemma. □

We will now show that traces are in one-to-one correspondence with the equivalence classes induced by the $\sim$ relation over the set of well-labeled transition sequences. We begin by showing that whenever a trace can be linearized into two sequences then they are equivalent. First a technical lemma: whenever the places on the incoming edges of a transition belong to the initial marking, this transition can shifted to the left of its current position.

Lemma 5.9. (*Left permutability of initial transitions*)
Let $\mathcal{T}$ be a trace. If $(\mathcal{S}, \xi, \mathcal{S}') \in \mathcal{S}(\mathcal{T})$ for $\xi = (t, \bullet x, x\bullet)$, and $\bullet x \subseteq \bullet\mathcal{T}$, then $(\xi, \mathcal{S}, \mathcal{S}') \in \mathcal{S}(\mathcal{T})$ and $(\mathcal{S}, \xi, \mathcal{S}') \sim (\xi, \mathcal{S}, \mathcal{S}')$.

Proof. The proof proceeds by induction on the structure of $\mathcal{S}$. The result holds trivially if $\mathcal{S} = \cdot$. If $\mathcal{S} = \mathcal{S}'', \xi''$, we show by induction on $\mathcal{S}'$ that $(\mathcal{S}'', \xi, \xi'', \mathcal{S}') \in \mathcal{S}(\mathcal{T})$, which allows us to appeal to the main induction hypothesis to show that $(\xi, \mathcal{S}, \mathcal{S}') \in \mathcal{S}(\mathcal{T})$ and $(\mathcal{S}, \xi, \mathcal{S}') \sim (\xi, \mathcal{S}, \mathcal{S}')$. □

This property is used in the proof of the anticipated lemma linking transition sequences underlying the same trace.

Lemma 5.10. (*Transition sequences underlying the same trace are equivalent*)
If $\mathcal{T}$ is a trace and $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{S}(\mathcal{T})$, then $\mathcal{S}_1 \sim \mathcal{S}_2$.

Proof. Let $\mathcal{T} = (\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E})$. We proceed by induction on the size of $\bar{\boldsymbol{T}}$. If $\bar{\boldsymbol{T}} = \varnothing$, then it must be the case that $\mathcal{S}_1 = \mathcal{S}_2 = \cdot$, which clearly satisfies the desired relation.

Assume then that $\mathcal{S}_1 = \xi_1, \mathcal{S}_1'$ and $\mathcal{S}_2 = \xi_2, \mathcal{S}_2'$. If $\xi_1 = \xi_2$, then $\mathcal{S}_1' \sim \mathcal{S}_2'$ by induction hypothesis since $\mathcal{S}_1'$ and $\mathcal{S}_2'$ must be defined on the same subtrace of $\mathcal{T}$. From this, we easily obtain the desired result.

Otherwise, $\mathcal{S}_1 = \xi_1, \mathcal{S}_1', \xi_2, \mathcal{S}_1''$ and $\mathcal{S}_2 = \xi_2, \mathcal{S}_2', \xi_1, \mathcal{S}_2''$. By lemma 5.9, we can conclude that $(\xi_2, \xi_1, \mathcal{S}_1', \mathcal{S}_1''), (\xi_1, \xi_2, \mathcal{S}_2', \mathcal{S}_2'') \in \mathcal{S}(\mathcal{T})$ and moreover $\mathcal{S}_1 \sim (\xi_2, \xi_1, \mathcal{S}_1', \mathcal{S}_1'')$ and $\mathcal{S}_2 \sim (\xi_1, \xi_2, \mathcal{S}_2', \mathcal{S}_2'')$. It is easy to show that $(\mathcal{S}_1', \mathcal{S}_1'')$ and $(\mathcal{S}_2', \mathcal{S}_2'')$ are defined on the same subtrace of $\mathcal{T}$. Therefore, by induction hypothesis, $(\mathcal{S}_1', \mathcal{S}_1'') \sim (\mathcal{S}_2', \mathcal{S}_2'')$ from which we deduce that $(\xi_2, \xi_1, \mathcal{S}_1', \mathcal{S}_1'') \sim (\xi_1, \xi_2, \mathcal{S}_2', \mathcal{S}_2'')$ and therefore $\mathcal{S}_1 \sim \mathcal{S}_2$ by the transitivity of $\sim$. □

We now prove the reverse inclusion by showing that equivalent execution sequences produce isomorphic traces. We will rely on the following lemma about well-labeled exchanges.

Lemma 5.11. (*Trace isomorphism for well-labeled exchanges*)
Given traces $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *such that* $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}_i$ *(for* $i = 1, 2$*), if* $\mathcal{S}_1 \sim_0 \mathcal{S}_2$ *then* $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}_1)$ *is isomorphic to* $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}_1)$.

Proof. By definition, $\mathcal{S}_1 = (\mathcal{S}, \xi, \xi', \mathcal{S}')$ and $\mathcal{S}_2 = (\mathcal{S}, \xi', \xi, \mathcal{S}')$ with $\xi = (t, \bullet x, x\bullet)$ and $\xi' = (t', \bullet x', x'\bullet)$. The proof proceeds by induction on $\mathcal{S}'$.

First, assume that $\mathcal{S}' = \cdot$. We expand $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$ as $(\bar{\boldsymbol{P}}, \bar{\boldsymbol{T}}, \mathbf{E})$. Then, by definition

$$\begin{aligned}
\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}, \xi, \xi') = (&\bar{\boldsymbol{P}} \cup (x\bullet{:}t\bullet) \cup (x'\bullet{:}t'\bullet), \\
&\bar{\boldsymbol{T}} \cup \{(y_1{:}t), (y_1'{:}t')\}, \\
&\mathbf{E} \cup \{(y_1{:}t) \mapsto ((\bullet x{:}\bullet t), (x\bullet{:}t\bullet)), (y_1'{:}t') \mapsto ((\bullet x'{:}\bullet t'), (x'\bullet{:}t'\bullet))\})
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}, \xi', \xi) = (&\bar{\boldsymbol{P}} \cup (x'\bullet{:}t'\bullet) \cup (x\bullet{:}t\bullet), \\
&\bar{\boldsymbol{T}} \cup \{(y_2'{:}t'), (y_2{:}t)\}, \\
&\mathbf{E} \cup \{(y_2'{:}t') \mapsto ((\bullet x'{:}\bullet t'), (x'\bullet{:}t'\bullet)), (y_2{:}t) \mapsto ((\bullet x{:}\bullet t), (x\bullet{:}t\bullet))\}).
\end{aligned}$$

Observe that $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}, \xi, \xi')$ and $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}, \xi', \xi)$ differ only by the name of the variables y_1, y_1' versus y_2, y_2'. They are therefore clearly isomorphic.

The inductive case, in which $\mathcal{S}' = (\mathcal{S}'', \xi)$, follows trivially by construction. □

This property extends naturally to equivalent well-labeled traces.

Lemma 5.12. (*Trace isomorphism for equivalent transition sequences*)
Given traces $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *such that* $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}_i$ *(for* $i = 1, 2$*), if* $\mathcal{S}_1 \sim \mathcal{S}_2$ *then* $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}_1)$ *is isomorphic to* $\mathcal{T}_{\boldsymbol{M}}(\mathcal{S}_1)$.

Proof. By Lemma 5.3, there are transition sequences $\mathcal{S}_0', \ldots, \mathcal{S}_n'$ such that $\mathcal{S}_1 = \mathcal{S}_0'$, $\mathcal{S}_2 = \mathcal{S}_n'$, and for $i = 1..n$, $\mathcal{S}_{i-1}' \sim_0 \mathcal{S}_i'$. Moreover, by an iterated application of Lemma 5.2, we have that $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}_i'$ for $i = 1..n$.

Given these observations, the main part of this proof proceeds by induction over n, using Lemma 5.11 in the inductive case. □

5.3 Sequential CLF Representation

We write $\ulcorner o \urcorner$ for the CLF representation of object entity o. Different entities have different representations. Whenever the same entity has several representations (for example sets of labels), we distinguish them as $\ulcorner o \urcorner^{(i)}$ where i is a progressive number. In some occasions, we write $\ulcorner o \urcorner^{o'}$ for the encoding of an object o parameterized by (the representation of) another object o'.

5.3.1 Representation of Petri nets

Let $N = (P, T, E)$ be a Petri net. We define the CLF representation of N, written $\ulcorner N \urcorner$, as follows:

$$\ulcorner N \urcorner = \ulcorner P \urcorner, \ulcorner E \urcorner$$

where $\ulcorner P \urcorner$ is in turn defined as:

$$\begin{array}{lcl} \ulcorner \varnothing \urcorner & = & \mathtt{place} : \mathsf{type}, \\ & & \mathtt{tok} : \mathtt{place} \rightarrow \mathsf{type} \\ \ulcorner P \cup \{p\} \urcorner & = & \ulcorner P \urcorner,\ p : \mathtt{place} \end{array}$$

and $\ulcorner E \urcorner$ is given by the following definitions:

$$\begin{array}{lcl} \ulcorner \varnothing \urcorner & = & \cdot \\ \ulcorner E \cup \{t \mapsto (\bullet t, t\bullet)\} \urcorner & = & \ulcorner E \urcorner,\ t : \ulcorner \bullet t \urcorner^{C} \quad \text{where } C = \{\, \ulcorner t\bullet \urcorner \,\} \end{array}$$

The encodings $\ulcorner t\bullet \urcorner$ and $\ulcorner \bullet t \urcorner^{C}$ are respectively defined as:

$$\begin{array}{lcl|lcl} \ulcorner \wr\int \urcorner & = & 1 & \ulcorner \wr\int \urcorner^{C} & = & C \\ \ulcorner \underline{t\bullet \uplus \wr p \int} \urcorner & = & \ulcorner \underline{t\bullet} \urcorner \otimes \mathtt{tok}\, p & \ulcorner \underline{\bullet t \uplus \wr p \int} \urcorner^{C} & = & \ulcorner \underline{\bullet t} \urcorner^{(\mathtt{tok}\, p \multimap C)} \end{array}$$

where we recall that the notation $\underline{m}$ indicates that the multiset m is viewed as a sequence with respect to some canonical order.

The application of these definition to the producer/consumer example from Figure 1 produces the following CLF signature:

$$\begin{array}{lcl} \mathsf{place} & : & \mathsf{type}. \\ \mathsf{tok} & : & \mathsf{place} \rightarrow \mathsf{type}. \end{array}$$

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl} \mathsf{r} & : & \mathsf{place}. & \mathsf{n} & : & \mathsf{place}. & \mathsf{c} & : & \mathsf{place}. \\ \mathsf{p} & : & \mathsf{place}. & \mathsf{b} & : & \mathsf{place}. & \mathsf{a} & : & \mathsf{place}. \end{array}$$

$$\begin{array}{lcl} \mathsf{t\_p} & : & \mathsf{tok\ p} \multimap \{1 \otimes \mathsf{tok\ r}\}. \\ \mathsf{t\_r} & : & \mathsf{tok\ r} \multimap \{1 \otimes \mathsf{tok\ p} \otimes \mathsf{tok\ b} \otimes \mathsf{tok\ n}\}. \\ \mathsf{t\_a} & : & \mathsf{tok\ b} \multimap \mathsf{tok\ b} \multimap \mathsf{tok\ a} \multimap \{1 \otimes \mathsf{tok\ c}\}. \\ \mathsf{t\_c} & : & \mathsf{tok\ c} \multimap \{1 \otimes \mathsf{tok\ a}\}. \end{array}$$

It should be observed how the encoding of the pre- and post-conditions of a transition are not symmetric: the former is modeled as iterated linear implications while the latter as an asynchronous formula inside a monad. It would be tempting to represent both using the monadic encoding, which is akin to the way Petri nets are traditionally rendered in linear logic [Cer95, MOM91, GG90]. For example, transition t_a would be represented as

$$\mathsf{t\_a} \ : \ \{1 \otimes \mathsf{tok\ b} \otimes \mathsf{tok\ b} \otimes \mathsf{tok\ a}\} \multimap \{1 \otimes \mathsf{tok\ c}\}.$$

While this is not incorrect, the behavior of this declaration is not what we would expect: it is applicable not only in a linear context containing two declarations of type tok b and one of type tok a, but also to any context that can be transformed (possibly by the application of other rules) into a context with these characteristics. In summary, our representation forces the execution of individual rules, while the alternative encoding allows its execution to be preceded by an arbitrary computation.

5.3.2 Representation of Markings

A marking $\boldsymbol{M}$ has two representations: one is simply a linear context consisting of all the places in $\boldsymbol{M}$ indexed by their label. We write it $\ulcorner \boldsymbol{M} \urcorner^{(1)}$.

$$\begin{array}{lcl} \ulcorner \varnothing \urcorner^{(1)} & = & \cdot \\ \ulcorner \boldsymbol{M} \cup (x,p) \urcorner^{(1)} & = & \ulcorner \boldsymbol{M} \urcorner^{(1)},\ x \hat{:} \mathtt{tok}\, p \end{array}$$

For example, the representation $\ulcorner \boldsymbol{M}_{pc1} \urcorner^{(1)}$ of the producer/consumer net marking $\boldsymbol{M}_{pc1}$ in Figure 1 is given by:

$$\begin{array}{llll} r_1 \hat{:} \mathsf{tok\ r.} & n_1 \hat{:} \mathsf{tok\ n.} & b_1 \hat{:} \mathsf{tok\ b.} & b_3 \hat{:} \mathsf{tok\ b.} \\ a_1 \hat{:} \mathsf{tok\ a.} & n_2 \hat{:} \mathsf{tok\ n.} & b_2 \hat{:} \mathsf{tok\ b.} & \end{array}$$

The second representation of a marking $\boldsymbol{M}$ consists of a pair whose first component is obtained by tensoring all the labels in $\boldsymbol{M}$. The second component is the tensor of the representation of the places in $\boldsymbol{M}$. We denote this encoding as $\ulcorner \boldsymbol{M} \urcorner^{(2)}$, which we will often expand as $(\ulcorner \boldsymbol{M} \urcorner^{(2')}, \ulcorner \boldsymbol{M} \urcorner^{(2'')})$. The two components are positionally synchronized with respect to $\boldsymbol{M}$: we achieve this by referring to the canonical ordering of this multiset, which we denoted $\underline{\boldsymbol{M}}$.

$$\begin{array}{lcl} \ulcorner \varnothing \urcorner^{(2)} & = & (1,1) \\ \ulcorner \underline{\boldsymbol{M} \cup (x,p)} \urcorner^{(2)} & = & (\ulcorner \underline{\boldsymbol{M}} \urcorner^{(2')} \otimes x,\ \ulcorner \underline{\boldsymbol{M}} \urcorner^{(2'')} \otimes \mathtt{tok}\, p) \end{array}$$

This second representation function yields the following pair when applied to the second marking $\boldsymbol{M}_{pc2}$ for the producer/consumer net (displayed in Figure 2):

$$\begin{array}{lllllllllllllllll} \ulcorner \boldsymbol{M}_{pc2} \urcorner^{(2)} = (& 1 & \otimes & r_2 & \otimes & c_1 & \otimes & n_1 & \otimes & n_2 & \otimes & n_3 & \otimes & b_3 & \otimes & b_4, \\ & 1 & \otimes & \mathsf{tok\ r} & \otimes & \mathsf{tok\ c} & \otimes & \mathsf{tok\ n} & \otimes & \mathsf{tok\ n} & \otimes & \mathsf{tok\ n} & \otimes & \mathsf{tok\ b} & \otimes & \mathsf{tok\ b}) \end{array}$$

It should be observed that this representation, and in particular the presence of labels, is a direct implementation of the "individual token approach" to Petri nets [BMMS98, BM00]. This derives from the fact that CLF, like most logical frameworks, forces every assumption to have a unique name. An extension of CLF with *proof irrelevance* [Pfe01a] would avoid this artificiality, allowing a direct representation of the "collective token philosophy". We intend to investigate this possibility as future work.

5.3.3 Representation of Execution Sequences

Let $\boldsymbol{M}$ and $\boldsymbol{M'}$ be markings and $S = \ulcorner \boldsymbol{M'} \urcorner^{(2')}$. Given the execution sequence $\mathcal{S}$ such that $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$, we denote the representation of $\mathcal{S}$ with respect to S as $\ulcorner \mathcal{S} \urcorner^{S}$. It is defined as follows:

$$\begin{array}{lcl} \ulcorner . \urcorner^{S} & = & S \\ \ulcorner (t, \bullet x, x\bullet), \mathcal{S} \urcorner^{S} & = & \mathsf{let}\ \ulcorner x\bullet \urcorner = \ulcorner \bullet x \urcorner^{t}\ \mathsf{in}\ \ulcorner \mathcal{S} \urcorner^{S} \end{array}$$

where the encodings $\ulcorner x\bullet \urcorner$ and $\ulcorner \bullet x \urcorner^{R}$ of the sets of variables $\bullet x$ and $x\bullet$ are defined as follows, respectively:

$$\begin{array}{lcl|lcl} \ulcorner \varnothing \urcorner & = & 1 & \ulcorner \varnothing \urcorner^{R} & = & R \\ \ulcorner x\bullet \cup \{x\} \urcorner & = & \ulcorner x\bullet \urcorner \otimes x & \ulcorner \bullet x \cup \{x\} \urcorner^{R} & = & \ulcorner \bullet x \urcorner^{R\,\hat{}\,x} \end{array}$$

We shall require that these encodings be positionally synchronized with $\ulcorner t \urcorner$ so that the labels in $\ulcorner \bullet x \urcorner^t$ match the places in the consequent of $\ulcorner t \urcorner$ and the labels in $\ulcorner x\bullet \urcorner$ are in the same order as the corresponding place representation in the antecedent of $\ulcorner t \urcorner$.

On the basis of these definitions, the CLF representation of the execution sequence $\mathcal{S}_{pc}$ with respect to $\ulcorner \boldsymbol{M}_{pc2} \urcorner^{(2')}$ given in Section 5.2.1 for the producer/consumer example has the following form:

$$\begin{array}{rlrcll} \ulcorner \mathcal{S}_{pc} \urcorner^{\ulcorner \boldsymbol{M}_{pc2} \urcorner^{(2')}} = & \mathsf{let} & 1 \otimes n_3 \otimes b_4 \otimes p_1 & = & \mathsf{t\_r}\,^\wedge r_1 & \mathsf{in} \\ & \mathsf{let} & 1 \otimes r_2 & = & \mathsf{t\_p}\,^\wedge p_1 & \mathsf{in} \\ & \mathsf{let} & 1 \otimes c_1 & = & \mathsf{t\_a}\,^\wedge a_1\,^\wedge b_1\,^\wedge b_2 & \mathsf{in} \\ & \multicolumn{5}{l}{1 \otimes r_2 \otimes c_1 \otimes n_1 \otimes n_2 \otimes n_3 \otimes b_3 \otimes b_4} \end{array}$$

5.3.4 Adequacy Theorems

We will now show that our encoding correctly captures the definition of Petri nets given in Section 5.2. More precisely, we will concentrate on the relation between transition sequences and canonical expressions derivable in our setting. We will first verify that the encoding of a transition sequence between two markings is a typable expression relative to the representation of these two markings.

Lemma 5.13. (*Soundness of the sequential representation of Petri nets*)
Let N be a Petri net and $\boldsymbol{M}, \boldsymbol{M'}$ be two markings over N. If $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$, *then there is a derivation $\mathcal{E}$ of*

$$\cdot; \ulcorner \boldsymbol{M} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2'')}$$

Proof. The proof proceeds by induction over $\mathcal{S}$. If $\mathcal{S}$ is the empty execution sequence, then $\boldsymbol{M} = \boldsymbol{M'}$. Since $\ulcorner \boldsymbol{M} \urcorner^{(2)}$ is positionally synchronized, it is easy to prove that the judgment

$$\cdot; \ulcorner \boldsymbol{M} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \boldsymbol{M} \urcorner^{(2')} \leftarrow \ulcorner \boldsymbol{M} \urcorner^{(2'')}$$

is derivable.

Assume now that $\mathcal{S} = (t, \bullet x, x\bullet), \mathcal{S}'$. Then, by definition of execution sequence,

$$\boldsymbol{M} \rhd_N \boldsymbol{M''} \text{ by } (t, \bullet x, x\bullet) \qquad \text{and} \qquad \boldsymbol{M''} \rhd_N^* \boldsymbol{M'} \text{ by } \mathcal{S}'$$

for some marking $\boldsymbol{M''}$. We then have that $\boldsymbol{M''} = (\boldsymbol{M} - (\bullet x{:}\bullet t)) \uplus (x\bullet{:}t\bullet)$.

By induction hypothesis, there is a derivation $\mathcal{E}'$ of

$$\cdot; \ulcorner \boldsymbol{M''} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2'')}$$

Observe that

$$\ulcorner \boldsymbol{M''} \urcorner^{(1)} = \Delta, \ulcorner (x\bullet{:}t\bullet) \urcorner^{(1)} \qquad \text{where } \Delta \text{ is s.t.} \qquad \ulcorner \boldsymbol{M} \urcorner^{(1)} = \Delta, \ulcorner (\bullet x{:}\bullet t) \urcorner^{(1)}$$

Now, $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} = \ulcorner (t, \bullet x, x\bullet), \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} = \mathsf{let}\ \ulcorner x\bullet \urcorner = \ulcorner \bullet x \urcorner^t\ \mathsf{in}\ \ulcorner \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}}$. Thanks to our synchronization assumptions on the definition of $\ulcorner x\bullet \urcorner$ and $\ulcorner \bullet x \urcorner^t$, it is easy to show that there is a CLF derivation $\mathcal{R}$ of $\cdot; \ulcorner (\bullet x{:}\bullet t) \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \bullet x \urcorner^t \Rightarrow \{\ulcorner t\bullet \urcorner\}$. Once we observe that the pattern abbreviation $\ulcorner x\bullet \urcorner : \ulcorner t\bullet \urcorner$ expands to $\ulcorner (x\bullet{:}t\bullet)\urcorner^{(1)}$, we simply use rule $\{\}\mathbf{E}$ to construct the desired derivation of $\cdot;\ulcorner \boldsymbol{M}\urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'}\urcorner^{(2'')}} \leftarrow \ulcorner \boldsymbol{M'}\urcorner^{(2'')}$. □

Although proving the adequacy of this encoding when execution sequences are extended from the left is relatively easy, as shown in the above proof, handling the dual extension is complicated and requires a procedure akin to cut-elimination for a sequent formulation of CLF.

It is easy to validate the above lemma on our running example: the following judgment is derivable

$$\cdot;\ulcorner \boldsymbol{M}_{pc1}\urcorner^{(1)} \vdash_{\ulcorner N_{pc} \urcorner} \ulcorner \mathcal{S}_{pc} \urcorner^{\ulcorner \boldsymbol{M}_{pc2}\urcorner^{(2')}} \leftarrow \ulcorner \boldsymbol{M}_{pc2}\urcorner^{(2'')}$$

We now show the complementary result: any well-typed CLF expression relative to two markings is the encoding of some transition sequence between them.

Lemma 5.14. (*Completeness of the sequential representation of Petri nets*)
Let N be a Petri net and $\boldsymbol{M}, \boldsymbol{M'}$ be two markings over N. If $\mathcal{E} :: \cdot;\ulcorner \boldsymbol{M}\urcorner^{(1)} \vdash_{\ulcorner N \urcorner} E \leftarrow \ulcorner \boldsymbol{M'}\urcorner^{(2'')}$, then there is an execution sequence $\mathcal{S}$ such that

$$\boldsymbol{M} \rhd^*_N \boldsymbol{M'} \text{ by } \mathcal{S}$$

and $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'}\urcorner^{(2')}} \equiv E$.

Proof. The proof proceeds by induction on the structure of $\mathcal{E}$. By inversion, there are two cases to examine.

1. Let us first consider the situation where $\mathcal{E}$ has rule $\Leftarrow\leftarrow$ as its last inference:

$$\mathcal{E} = \cfrac{\begin{array}{c}\mathcal{E}' \\ \cdot;\ulcorner \boldsymbol{M}\urcorner^{(1)} \vdash_{\ulcorner N \urcorner} M \Leftarrow \ulcorner \boldsymbol{M'}\urcorner^{(2'')}\end{array}}{\cdot;\ulcorner \boldsymbol{M}\urcorner^{(1)} \vdash_{\ulcorner N \urcorner} M \leftarrow \ulcorner \boldsymbol{M'}\urcorner^{(2'')}} \Leftarrow\leftarrow$$

   where $E = M$.

   We prove by induction on the structure of $\ulcorner \boldsymbol{M'}\urcorner^{(2'')}$ that if $\cdot;\Delta \vdash_{\ulcorner N \urcorner} M \Leftarrow \ulcorner \boldsymbol{M'}\urcorner^{(2'')}$, then $\Delta \equiv \ulcorner \boldsymbol{M'}\urcorner^{(1)}$ and that $M \equiv \ulcorner . \urcorner^{\ulcorner \boldsymbol{M}\urcorner^{(2')}}$. If $\boldsymbol{M'} = \varnothing$, then $\ulcorner \boldsymbol{M'}\urcorner^{(2'')} \equiv 1$. By inversion on rule $\mathbf{1I}$, we deduce that $M \equiv 1$ and $\ulcorner \boldsymbol{M'}\urcorner^{(1)} \equiv \cdot$, which is exactly what we are looking for.

   Assume now that $\boldsymbol{M'} = \boldsymbol{M^*}, (x{:}p)$ so that $\ulcorner \boldsymbol{M'}\urcorner^{(2'')} = \ulcorner \boldsymbol{M^*}\urcorner^{(2'')} \otimes \texttt{tok}\, p$. Then, by inversion on rule $\otimes\mathbf{I}$, we can rewrite $\mathcal{E}'$ as

$$\cfrac{\begin{array}{cc}\mathcal{E}'_1 & \mathcal{E}'_2 \\ \cdot;\Delta_1 \vdash_{\ulcorner N \urcorner} M_1 \Leftarrow \ulcorner \boldsymbol{M^*}\urcorner^{(2'')} & \cdot;\Delta_2 \vdash_{\ulcorner N \urcorner} M_2 \Leftarrow \texttt{tok}\, p\end{array}}{\cdot;\Delta_1, \Delta_2 \vdash_{\ulcorner N \urcorner} M_1 \otimes M_2 \Leftarrow \ulcorner \boldsymbol{M^*}\urcorner^{(2'')} \otimes \texttt{tok}\, p} \otimes\mathbf{I}$$

   By induction hypothesis on $\mathcal{E}_1$, $\Delta_1 \equiv \ulcorner \boldsymbol{M^*}\urcorner^{(1)}$ and $M \equiv \ulcorner . \urcorner^{\ulcorner \boldsymbol{M^*}\urcorner^{(2')}}$. We only need to show that $\Delta_2 \equiv x{:}\texttt{tok}\, p$ and $M_2 \equiv x$. Since all positive occurrences of atomic types of the form $\mathtt{tok}\,p$ appear within a monad in $\ulcorner N \urcorner$, only rule $\mathbf{x}$ is applicable, which precisely satisfies our requirements.

The result we just proved allows us to deduce that $\boldsymbol{M} = \boldsymbol{M'}$ in $\mathcal{E}$. Therefore, it must be the case that $\mathcal{S} = \cdot$, so that $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} \equiv M$.

2. Assume now that the last rule applied in $\mathcal{E}$ is

$$\dfrac{\begin{array}{cc} \mathcal{E}_1 & \mathcal{E}_2 \\ \cdot; \Delta_1 \vdash_{\ulcorner N \urcorner} R \Rightarrow \{S\} & \cdot; \Delta_2; p \mathbin{\hat{:}} S \vdash_{\ulcorner N \urcorner} E' \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2')} \end{array}}{\cdot; \Delta_1, \Delta_2 \vdash_{\ulcorner N \urcorner} \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E' \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2'')}}\ \{\}\mathbf{E}$$

where $\ulcorner \boldsymbol{M} \urcorner^{(1)} \equiv (\Delta_1, \Delta_2)$ and $E \equiv (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E')$.

By inspection over $\ulcorner N \urcorner$, a type of the form $\{S\}$ appears only in the declaration $\ulcorner t \urcorner$ for some transition t, and it cannot be constructed. Therefore, it must be the case that $S \equiv \ulcorner t\bullet \urcorner$. We then prove by induction over $\bullet t$ that $\Delta_1 \equiv (\bullet x{:}\bullet t)$ for appropriate variables $\bullet x$, and that $R \equiv \ulcorner \bullet x \urcorner^t$.

By inversion on $\mathcal{E}_2$, the right premise of $\mathcal{E}$ reduces to $\cdot; \Delta_2, (x\bullet{:}t\bullet) \vdash_{\ulcorner N \urcorner} E' \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2'')}$ for appropriate variables $x\bullet$. The context $\Delta_2, (x\bullet{:}t\bullet)$ corresponds therefore to the representation of a marking $\boldsymbol{M^*}$ such that $\boldsymbol{M^*} = (\boldsymbol{M} - (\bullet x{:}\bullet t)) \uplus (x\bullet{:}t\bullet)$. By induction hypothesis, there is a transition sequence $\mathcal{S}'$ such that $\boldsymbol{M^*} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}'$ and $\ulcorner \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} \equiv E'$.

If we now define $\mathcal{S} = (t, \bullet x, x\bullet), \mathcal{S}'$, we obtain that $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$ and $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}} \equiv E$

This concludes our proof. □

We can finally put these two results together in the following adequacy theorem for our sequential CLF representation of Petri nets.

Theorem 5.15. (*Adequacy of the sequential representation of Petri nets*)
Let N be a Petri net and $\boldsymbol{M}, \boldsymbol{M'}$ be two markings over N. There is a bijection between execution sequences $\mathcal{S}$ such that

$$\boldsymbol{M} \rhd_N^* \boldsymbol{M'} \text{ by } \mathcal{S}$$

and terms E such that the judgment

$$\cdot; \ulcorner \boldsymbol{M} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} E \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2'')}$$

is derivable in CLF.

Proof. This is a simple corollary of Lemmas 5.13 and 5.14. □

5.4 Concurrent CLF Representation

Another way to express the behavior of a Petri net in CLF is to give a representation of traces and show its adequacy. Rather than presenting a direct encoding of these objects, we will exploit the meta-theory that we have developed in Section 5.2 to relate traces and transition sequences. We will indirectly represent a trace through one of its underlying transition sequences, in such a way that the definitional equality of CLF make the choice of the actual representative irrelevant.

5.4.1 Representation of Traces

We will keep the encoding of Petri nets and markings unchanged from Section 5.3. Recall that a trace $\mathcal{T}$ over a Petri net N relates the initial marking ${}^\bullet\mathcal{T}$ of $\mathcal{T}$ to its final marking $\mathcal{T}^\bullet$. We will rely on the encodings $\ulcorner {}^\bullet\mathcal{T} \urcorner^{(1)}$ and $\ulcorner \mathcal{T}^\bullet \urcorner^{(2)} = (\ulcorner \mathcal{T}^\bullet \urcorner^{(2')}, \ulcorner \mathcal{T}^\bullet \urcorner^{(2'')})$ defined in Section 5.3.2.

We denote the representation of a trace $\mathcal{T}$ as $\ulcorner \mathcal{T} \urcorner$. It is defined as $\ulcorner \mathcal{S} \urcorner^S$ for an arbitrary execution sequence $\mathcal{S} \in \mathcal{S}(\mathcal{T})$, where $S = \ulcorner \mathcal{T}^\bullet \urcorner^{(2')}$ as defined in Section 5.3. Therefore, the representation $\ulcorner \mathcal{T}_{pc} \urcorner$ of the producer/consumer net shown in Figure 3 can be defined as $\ulcorner \mathcal{S}_{pc} \urcorner^S$ where $S = \ulcorner \mathcal{T}_{pc}^\bullet \urcorner^{(2')}$. The attentive reader may object that a different pick $\mathcal{S}'$ of the transition sequence chosen from $\mathcal{S}(\mathcal{T})$ will yield a different CLF expression, so that the encoding of a trace $\mathcal{T}$ is not well-defined. While we acknowledge this observation, we will show that the representations of any two transition sequences in $\mathcal{S}(\mathcal{T})$ are equal modulo $=_c$, and will therefore be indistinguishable as CLF objects.

We know by Lemma 5.10 that if two transition sequences $\mathcal{S}$ and $\mathcal{S}'$ underly the same trace $\mathcal{T}$, then they are equivalent modulo $\sim$. We will therefore prove that the CLF encoding of two such traces are equal modulo $=_c$. In order to show this result, we will need the following lemma that states that this property holds of well-labeled exchanges.

Lemma 5.16. (*Well-labeled exchanges have equal CLF representations*)
Let $\mathcal{S}$ and $\mathcal{S}'$ be two executions sequences and $\boldsymbol{M}$ a marking over a Petri net N. If $\mathcal{S} \sim_0 \mathcal{S}'$, then $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}} =_c \ulcorner \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$.

Proof. Let $\mathcal{S} = \bar{\mathcal{S}}, \xi, \xi', \mathcal{S}'$ and $\mathcal{S}' = \bar{\mathcal{S}}, \xi', \xi, \mathcal{S}'$ with $\xi = (t, {}^\bullet x, x^\bullet)$ and $\xi' = (t', {}^\bullet x', x'^\bullet)$.

By definition, $\ulcorner \xi, \xi', \bar{\mathcal{S}}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}} = \mathsf{let}\ \ulcorner x^\bullet \urcorner = \ulcorner {}^\bullet x] \urcorner^t\ \mathsf{in}\ (\mathsf{let}\ \ulcorner x'^\bullet \urcorner = \ulcorner {}^\bullet x' \urcorner^{t'}\ \mathsf{in}\ \bar{E})$ and $\ulcorner \xi', \xi, \bar{\mathcal{S}}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}} = \mathsf{let}\ \ulcorner x'^\bullet \urcorner = \ulcorner {}^\bullet x' \urcorner^{t'}\ \mathsf{in}\ (\mathsf{let}\ \ulcorner x^\bullet \urcorner = \ulcorner {}^\bullet x \urcorner^t\ \mathsf{in}\ \bar{E})$ where $\bar{E} = \ulcorner \bar{\mathcal{S}}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$.

Since $\mathcal{S}$ and $\mathcal{S}'$ are well-labeled, we have that ${}^\bullet x \cap x'^\bullet = \varnothing$ and ${}^\bullet x' \cap x^\bullet = \varnothing$. By definition of execution sequence, we clearly have that $x^\bullet \cap x'^\bullet = \varnothing$. Therefore, $\ulcorner \xi, \xi', \bar{\mathcal{S}}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$ and $\ulcorner \xi', \xi, \bar{\mathcal{S}}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$ satisfy the constraints to applying the `let`-rule for definitional equality and

$$\mathsf{let}\ \ulcorner x^\bullet \urcorner = \ulcorner {}^\bullet x] \urcorner^t\ \mathsf{in}\ (\mathsf{let}\ \ulcorner x'^\bullet \urcorner = \ulcorner {}^\bullet x' \urcorner^{t'}\ \mathsf{in}\ \bar{E}) =_c \mathsf{let}\ \ulcorner x'^\bullet \urcorner = \ulcorner {}^\bullet x' \urcorner^{t'}\ \mathsf{in}\ (\mathsf{let}\ \ulcorner x^\bullet \urcorner = \ulcorner {}^\bullet x \urcorner^t\ \mathsf{in}\ \bar{E}).$$

where the witnessing concurrent context is $\epsilon = \mathsf{let}\ \ulcorner x'^\bullet \urcorner = \ulcorner {}^\bullet x' \urcorner^{t'}\ \mathsf{in}\ \_$.

Since $=_c$ is a congruence, a simple inductive argument shows that $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}} =_c \ulcorner \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$. □

The similar result for well-labeled equivalent transitions sequences then follows immediately.

Lemma 5.17. (*Well-labeled equivalent transition sequences have equal CLF representations*)
Let $\mathcal{S}$ and $\mathcal{S}'$ be two executions sequences and $\boldsymbol{M}$ a marking over a Petri net N. If $\mathcal{S} \sim \mathcal{S}'$, then $\ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}} =_c \ulcorner \mathcal{S}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$.

Proof. By Lemma 5.3, there are transition sequences $\mathcal{S}_0, \ldots, \mathcal{S}_n$ such that $\mathcal{S} = \mathcal{S}_0$, $\mathcal{S}' = \mathcal{S}_n$, and for $i = 1..n$, $\mathcal{S}_{i-1} \sim_0 \mathcal{S}_i$.

By the above Lemma 5.16, $\ulcorner\mathcal{S}_{i-1}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c \ulcorner\mathcal{S}_i\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')}$ for $i = 1..n$. Therefore, by the transitivity of $=_c$, $\ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c \ulcorner\mathcal{S}'\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')}$. □

We can now patch together the various parts of our argument to show that the representation of a trace is indeed well-defined.

Corollary 5.18. (*Representation of traces is well-defined*)
Let $\mathcal{T}$ be a trace over a Petri net N. Then $\ulcorner\mathcal{T}\urcorner$ is well-defined modulo $=_c$.

Proof. By Lemma 5.10, if $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{S}(\mathcal{T})$, then $\mathcal{S}_1 \sim \mathcal{S}_2$. By Lemma 5.17, for any $\boldsymbol{M}$, $\ulcorner\mathcal{S}_2\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c \ulcorner\mathcal{S}_2\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')}$, in particular for $\boldsymbol{M} = {}^\bullet\mathcal{T}$. □

Knowing that every transition sequence underlying a trace $\mathcal{T}$ is mapped to equal CLF expressions (modulo $=_c$) is however not sufficient: there may be CLF objects that are equal modulo $=_c$ to the representation of $\mathcal{T}$, but that are not the encoding of any transition sequence underlying it. We will now show that this option cannot materialize. In order to do so, we will prove that any such object must be the representation of some trace underlying $\mathcal{T}$, and therefore of a member of $\mathcal{S}(\mathcal{T})$.

Lemma 5.19. (*Completeness of the representation for well-labeled equivalent transition sequences*)
Let $\mathcal{S}$ be a well-labeled executions sequences and $\boldsymbol{M}$ a marking over a Petri net N. If $\ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c E$, then there is an execution sequence $\mathcal{S}'$ such that $\mathcal{S} \sim \mathcal{S}'$ and $E \equiv \ulcorner\mathcal{S}'\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')}$.

Proof. We proceed by induction on a derivation $\mathcal{E}$ of the judgment $\ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c E$. We need to distinguish two cases.

1. We first examine the situation where E is a monadic object M. Therefore,

$$\mathcal{E} = \cfrac{\begin{array}{c}\mathcal{E}' \\ \ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} = M\end{array}}{\ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c M}$$

   By definition, $\ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')}$ is a monadic object if and only if $\mathcal{S} = \cdot$, and furthermore this term reduces to $\ulcorner\boldsymbol{M}\urcorner^{(2')}$.

   A simple induction on the number of elements in $\boldsymbol{M}$ shows that M must be identical to $\ulcorner\boldsymbol{M}\urcorner^{(2')}$, hence $M \equiv \ulcorner\boldsymbol{M}\urcorner^{(2')}$.

2. The second option requires that both sides of $=_c$ be proper expressions. In order for this to be possible, it must be the case that $\mathcal{S} = (t, {}^\bullet x, x^\bullet), \hat{\mathcal{S}}$ for some transition t, so that $\ulcorner\mathcal{S}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} = \mathsf{let}\ \ulcorner x^\bullet\urcorner = \ulcorner{}^\bullet x\urcorner^t\ \mathsf{in}\ \ulcorner\hat{\mathcal{S}}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')}$. The last rule applied in $\mathcal{E}$ has therefore the form:

$$\mathcal{E} = \cfrac{\begin{array}{c}\mathcal{E}' \\ \ulcorner{}^\bullet x\urcorner^t = R\end{array} \qquad \begin{array}{c}\mathcal{E}'' \\ \ulcorner\hat{\mathcal{S}}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c \epsilon[\hat{E}]\end{array}}{\mathsf{let}\ \ulcorner x^\bullet\urcorner = \ulcorner{}^\bullet x\urcorner^t\ \mathsf{in}\ \ulcorner\hat{\mathcal{S}}\urcorner^{\ulcorner\boldsymbol{M}\urcorner(2')} =_c \epsilon[\mathsf{let}\ \ulcorner x^\bullet\urcorner = R\ \mathsf{in}\ \hat{E}]}$$

where $E = \epsilon[\mathsf{let}\ \ulcorner x\bullet \urcorner = R\ \mathsf{in}\ \hat{E}]$ for some concurrent context ϵ, atomic object R and expression $\hat{E}$.

By reasoning steps similar to the first part of this proof, we can deduce that $R \equiv \ulcorner \bullet x \urcorner^t$. By induction hypothesis on $\mathcal{E}''$, we also have that $\hat{\mathcal{S}} \sim \hat{\mathcal{S}}'$ and $\epsilon[\hat{E}] \equiv \ulcorner \hat{\mathcal{S}}' \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$. By induction on ϵ, we can partition $\hat{\mathcal{S}}'$ into two subsequences $\hat{\mathcal{S}}'_\epsilon$ and $\hat{\mathcal{S}}'_{\hat{E}}$ such that $\hat{\mathcal{S}}' = \hat{\mathcal{S}}'_\epsilon, \hat{\mathcal{S}}'_{\hat{E}}$, and $\hat{E} \equiv \ulcorner \hat{\mathcal{S}}'_{\hat{E}} \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$ and ϵ encodes $\hat{\mathcal{S}}'_\epsilon$ (with some abuse of notation, we may write $\epsilon \equiv \ulcorner \hat{\mathcal{S}}'_\epsilon \urcorner$-).

By the definition of $\ulcorner \_ \urcorner$-, it is clear that $\epsilon[\mathsf{let}\ \ulcorner x\bullet \urcorner = \ulcorner \bullet x \urcorner^t\ \mathsf{in}\ \hat{E}] \equiv \ulcorner \hat{\mathcal{S}}_\epsilon, (t, \bullet x, x\bullet), \hat{\mathcal{S}}'_{\hat{E}} \urcorner^{\ulcorner \boldsymbol{M} \urcorner^{(2')}}$. Therefore, we only need to prove that $\mathcal{S} \sim (\hat{\mathcal{S}}_\epsilon, (t, \bullet x, x\bullet), \hat{\mathcal{S}}'_{\hat{E}})$ (remember that $\mathcal{S} = (t, \bullet x, x\bullet), \hat{\mathcal{S}})$. By definition of execution sequence equivalence, this reduces to showing that $(\hat{\mathcal{S}}_\epsilon, (t, \bullet x, x\bullet), \hat{\mathcal{S}}'_{\hat{E}})$ is well-labeled.

A sequence is well-labeled if every label in it occurs at most once in a pre-condition or post-condition, and whenever it occurs in both, the transition mentioning it in its post-condition is to the left of the transition having it in its pre-conditions. The first requirement holds of $(\hat{\mathcal{S}}_\epsilon, (t, \bullet x, x\bullet), \hat{\mathcal{S}}'_{\hat{E}})$ since $\mathcal{S}$ is well-labeled and contains exactly the same transition witnesses. The second requirement derives from the constraints associated with the above rule. In particular, in order for this rule to be applicable, it must be the case that $\bullet x \cap \mathrm{BV}(\epsilon) = \emptyset$. Now, by definition, $\mathrm{BV}(\epsilon)$ collects all the post-conditions occurring in $\hat{\mathcal{S}}'_\epsilon$. Therefore $(\hat{\mathcal{S}}_\epsilon, (t, \bullet x, x\bullet), \hat{\mathcal{S}}'_{\hat{E}})$ is a well-labeled permutation of t $\mathcal{S}$ and therefore $\mathcal{S} \sim (\hat{\mathcal{S}}_\epsilon, (t, \bullet x, x\bullet), \hat{\mathcal{S}}'_{\hat{E}})$.

This concludes our proof. □

We now chain our findings together by showing that $\ulcorner \mathcal{T} \urcorner$ does not contain extraneous expressions.

Corollary 5.20. (*Completeness of the representation for traces*)
Let $\mathcal{T}$ be a trace over a Petri net N. If $E =_c \ulcorner \mathcal{T} \urcorner$, then $E \equiv \ulcorner \mathcal{T} \urcorner$.

Proof. By definition, $\ulcorner \mathcal{T} \urcorner \equiv \ulcorner \mathcal{S} \urcorner^S$ with $\mathcal{S} \in \mathcal{S}(\mathcal{T})$ and $S \equiv \ulcorner \bullet \mathcal{T} \urcorner^{(2')}$. By Lemma 5.19, $E \equiv \ulcorner \mathcal{S}' \urcorner^S$ for some $\mathcal{S}'$ such that $\mathcal{S} \sim \mathcal{S}'$. By Lemma 5.12, $\mathcal{S} \in \mathcal{S}(\mathcal{T})$ and so $E \equiv \ulcorner \mathcal{T} \urcorner$. □

5.4.2 Adequacy Theorems

The results we just unveiled give us simple means for proving the adequacy of the concurrent representation of Petri nets. We start with the soundness property: the encoding of a trace is a well-typed expression relative to the representation of its initial and final marking.

Lemma 5.21. (*Soundness of the concurrent representation of Petri nets*)
If $\mathcal{T}$ be a trace over a Petri net N, then there is a derivation $\mathcal{E}$ of

$$\cdot; \ulcorner \bullet \mathcal{T} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \mathcal{T} \urcorner \leftarrow \ulcorner \mathcal{T} \bullet \urcorner^{(2'')}.$$

Proof. By Corollary 5.7, for any derivation sequence $\mathcal{S} \in \mathcal{S}(\mathcal{T})$ we have that $\bullet\mathcal{T} \rhd_N^* \mathcal{T}\bullet$ by $\mathcal{S}$. By the Adequacy Lemma 5.13, we have that there is a derivation $\mathcal{E}$ of

$$\cdot; \ulcorner \bullet\mathcal{T} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} \ulcorner \mathcal{S} \urcorner^{S} \leftarrow \ulcorner \mathcal{T}\bullet \urcorner^{(2'')}$$

where $S \equiv \ulcorner \mathcal{T}\bullet \urcorner^{(2')}$. By the definition of $\ulcorner \mathcal{T} \urcorner$, this is exactly the derivation we want. $\square$

Moreover, any canonical expression relative to the encoding of the initial and final marking of a valid trace is the representation of this trace.

Lemma 5.22. (*Completeness of the concurrent representation of Petri nets*)
Let N be a Petri net and $\boldsymbol{M}, \boldsymbol{M'}$ be two markings over N. If $\mathcal{E} :: \cdot; \ulcorner \boldsymbol{M} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} E \leftarrow \ulcorner \boldsymbol{M'} \urcorner^{(2'')}$, then there is a trace $\mathcal{T}$ such that $\boldsymbol{M} = \bullet\mathcal{T}$, $\boldsymbol{M'} = \mathcal{T}\bullet$, and $E \equiv \ulcorner \mathcal{T} \urcorner$.

Proof. By the Adequacy Lemma 5.14, there is an execution sequence $\mathcal{S}$ such that $\boldsymbol{M} \rhd_N^* \boldsymbol{M'}$ by $\mathcal{S}$ and $E \equiv \ulcorner \mathcal{S} \urcorner^{\ulcorner \boldsymbol{M'} \urcorner^{(2')}}$.
Let $\mathcal{T} = \mathcal{T}_{\boldsymbol{M}}(\mathcal{S})$. By Corollary 5.6, $\mathcal{T}$ is a trace and $\bullet\mathcal{T} = \boldsymbol{M}$ and $\mathcal{T}\bullet = \boldsymbol{M'}$. $\square$

We bring these two results together in the following adequacy theorem.

Theorem 5.23. (*Adequacy of the concurrent representation of Petri nets*)
Let N be a Petri net. There is a bijection between traces $\mathcal{T}$ over and terms E such that the judgment

$$\cdot; \ulcorner \bullet\mathcal{T} \urcorner^{(1)} \vdash_{\ulcorner N \urcorner} E \leftarrow \ulcorner \mathcal{T}\bullet \urcorner^{(2'')}$$

is derivable in CLF.

Proof. This is a simple corollary of Lemmas 5.21 and 5.22. $\square$

5.5 Petri Nets in LLF

In this section, we will present an encoding of our example, the producer/consumer Petri net given in Figure 1, in LLF and compare it to the CLF representations obtained earlier. This formalization is based on [Cer95]. We remind the reader that LLF is the sublanguage of CLF that only allows Π, $\multimap$, $\&$ and $\top$ as type constructors and correspondingly limits the language of terms.

We will keep the representation of the antecedent of a transition unchanged with respect to our previous CLF encoding, but we need to apply deep alterations to the representation of the consequent since LLF does not embed any connective akin to $\otimes$. With linear implication as the only multiplicative connective at our disposal, we are forced to formalize transitions in a continuation-passing style: we introduce a control atom, that we call baton, that the representation of a transition acquires when it is fired and passes on to the next transition when it is done (like in a relay race).

The application of this idea to our example yields the LLF signature $\ulcorner N_{pc} \urcorner$ below, where we have occasionally written a type $A \multimap B$ as $B \circ\!\!- A$ as is often done in (linear) logic programming:

```
p     : type.
r     : type.
n     : type.
b     : type.
a     : type.
c     : type.
baton : type.

t_p : baton ⟜ p
            ⟜ (r ⊸ baton).
t_p : baton ⟜ r
            ⟜ (p ⊸
                 n ⊸
                 b ⊸ baton).

t_a : baton ⟜ a
            ⟜ b
            ⟜ b
            ⟜ (r ⊸ baton).
t_c : baton ⟜ c
            ⟜ (a ⊸ baton).
```

The behavior of CLF's $\otimes$ is implicitly recovered, in part, in the above encoding since right-nested linear implications can be curried: $A \multimap B \multimap C$ is logically equivalent to $A \otimes B \multimap C$. Writing our example in this way would however take us outside of LLF.

The initial marking $\boldsymbol{M}_{pc1}$ in Figure 1 can simply be flattened into the linear context of LLF (below, left). The treatment of the final marking $\boldsymbol{M}_{pc2}$ (see Figure 2) is however not as immediate: we represent it as if it were a transition (stop) that consumes all the tokens in the marking, and moreover that can be used at most once (below, right).

```
r1 ^: r.          stop ^: baton ⟜ r
n1 ^: n.                        ⟜ n
n2 ^: n.                        ⟜ n
b1 ^: b.                        ⟜ n
b2 ^: b.                        ⟜ b
b3 ^: b.                        ⟜ b
a1 ^: a.                        ⟜ c
```

We call the LLF encodings of these two markings $\ulcorner \boldsymbol{M}_{pc1} \urcorner$ and $\ulcorner \boldsymbol{M}_{pc2} \urcorner$, respectively.

Given this encoding, there is a one-to-one correspondence between firing sequences $\mathcal{S}$ such that $\boldsymbol{M}_{pc1} \rhd^*_{N_{pc}} \boldsymbol{M}_{pc2}$ by $\mathcal{S}$ and LLF proof-terms M such that

$$\cdot ; \ulcorner \boldsymbol{M}_{pc1} \urcorner, \ulcorner \boldsymbol{M}_{pc2} \urcorner \vdash^{LLF}_{\ulcorner N_{pc} \urcorner} M : \mathsf{baton}$$

is derivable.

The main difference between the CLF and the LLF representations of Petri nets is the threading of transition firing that we have implemented in the latter by means of the control atom baton. This encoding is inherently sequential. The permutability of independent transitions can only be established as a meta-theoretic relation between two proof-terms, a process that we have found to be overwhelming in general. The CLF representation does not force a control thread on the encoding of transitions. This, coupled with the equational theory of this new formalism, enables an implicit and faithful rendering of the permutability of independent Petri net transitions. Therefore, we claim that LLF is a stateful but inherently sequential language, while CLF natively supports a form of concurrency that seems to be extremely general.

A minor difference between the two encodings concerns the representation of markings, especially the final marking of an execution sequence. In LLF, we are bound to enter it as a special use-once rule. CLF supports a more direct encoding as a synchronous type in the right-hand side of the typing judgment. Altogether, we find the CLF representation more direct than what we could produce using LLF.

6 Specification of Security Protocol

Since their inception in the early 1960's [Pet62], Petri nets have been the object of many extensions aimed at increasing the abstraction level of a specification, at making them more expressive, and at using their modeling power for specific applications. In this section, we will analyze one such extension, the security protocol specification framework MSR [CDL$^+$99, Cer01b, Cer01a] and describe a CLF encoding for it.

Rather than the amorphous objects seen in Section 5, MSR's tokens are *facts*, *i.e.,* typed first-order ground atomic formulas over a term language of cryptographic messages. Transitions (called *rules* in MSR) become parametric in the inner structure of facts: variables can be matched at application time with the data carried by a fact, allowing in this way the same rule to apply in situations that differ by the value of some fields. Systems of this sort are known as *colored Petri nets* [Jen97]; they are also very closely related to various languages based on multiset rewriting such as *GAMMA* [BL93] (MSR actually stand for MultiSet Rewriting). MSR extends this model with a sophisticate type system and the possibility of creating fresh data dynamically. Moreover, it applies it to the specific domain of security protocols.

We describe relevant aspects of the syntax and operational behavior of MSR in Section 6.1. Then we will show in Section 6.2 how the basic CLF encoding for Petri nets is adapted to accommodate the many extensions present in MSR.

6.1 The Security Protocol Specification Language MSR

A security protocol describes the message exchange taking place between two or more agents, or *principals*, who wish to perform tasks such as establishing a secure channel to communicate confidential data, verifying each other's identity, etc. Messages are assumed to be sent over a public network, such as the Internet, and therefore run the risk of being intercepted and even fabricated by attackers. Cryptography is used to achieve virtual private channels over this public medium.

There has been a recent surge in formalisms for specifying security protocols, together with methods for proving their correctness. One such language is the Spi calculus [AG99], which specializes the π-calculus with dedicated cryptographic constructs. It can be expressed in CLF through a simple adaptation of the encoding presented in Section 3. In this section, we will sketch a CLF encoding for another protocol specification language: MSR [CDL$^+$99, Cer01b], a distant cousin of Petri nets.

MSR is a flexible framework for expressing security protocol and the linguistic infrastructure they rely on. Rather than describing the meta-language in its generality, we will concentrate on an instance tailored for the description of a specific protocol, the Needham-Schroeder public key protocol that we will use as an extended example. We will actually throw in a handful of additional constructs so that the reader can appreciate the expressive power of this formalism. More precisely, we introduce the syntax of MSR in Section 6.1.1, apply it to an example in Section 6.1.2, and conclude with the operational semantics of this language in Section 6.1.3.

6.1.1 Syntax

We start by defining the syntax of messages, which constitute the term language of the instance of MSR we are considering in this document. *Atomic messages* consist of principal names, cryptographic keys, and nonces (short-lived data used for authentication purposes). They are formally given by the following grammar:

$$\begin{array}{lrcll} \textit{Atomic messages:} & a & ::= & \mathsf{A} & \textit{(Principal)} \\ & & \mid & \mathsf{k} & \textit{(Key)} \\ & & \mid & \mathsf{n} & \textit{(Nonce)} \end{array}$$

With just a few exceptions, we will use the displayed identifiers, possibly adorned with subscripts or superscripts, to refer to objects of the associated syntactic class. For example, the symbol n_3 will generally identify a nonce. As an additional convention, we will use a serifed font (*e.g.,* n_3) to denote constants while reserving a generic font (*e.g.,* n_3) to denote objects that either are or may be variables (introduced shortly).

Messages are either constants, variables, the concatenation of two terms, or the result of encrypting a term with a key. We display syntax for encryption with both symmetric and asymmetric keys, although we will be using only the latter.

$$\begin{array}{lrcll} \textit{Messages:} & t & ::= & a & \textit{(Atomic messages)} \\ & & \mid & x & \textit{(Variables)} \\ & & \mid & t_1\, t_2 & \textit{(Concatenation)} \\ & & \mid & \{t\}_{\mathsf{k}} & \textit{(Symmetric-key encryption)} \\ & & \mid & \{\!\{t\}\!\}_{\mathsf{k}} & \textit{(Asymmetric-key encryption)} \end{array}$$

It should be observed that these declaration cast a layer of symbolic abstraction over the bit-strings that implement the messages of a security protocol. This approach, which can be found in almost all security protocol specifications and analysis environments, is known as the Dolev-Yao abstraction [NS78, DY83].

An *elementary term* is either a constant or a variable:

$$\begin{array}{lrcll} \textit{Elementary terms} & e & ::= & a & \textit{(Constants)} \\ & & \mid & x & \textit{(Variables)} \end{array}$$

Predicates may take multiple arguments. Consequently, we introduce syntax for tuples of messages:

$$\begin{array}{lrcll} \textit{Message tuples} & \vec{t} & ::= & \cdot & \textit{(Empty tuple)} \\ & & \mid & t,\, \vec{t} & \textit{(Tuple extension)} \end{array}$$

In MSR, every object has a type drawn from the theory of dependent types with subsorting [Cer01a]. In this paper, we will use the following layout:

$$\begin{array}{lrcll} \textit{Types:} & \tau & ::= & \mathsf{principal} & \textit{(Principals)} \\ & & \mid & \mathsf{nonce} & \textit{(Nonces)} \\ & & \mid & \mathsf{shK}\ A\ B & \textit{(Shared keys)} \\ & & \mid & \mathsf{pubK}\ A & \textit{(Public keys)} \\ & & \mid & \mathsf{privK}\ k & \textit{(Private keys)} \\ & & \mid & \mathsf{msg} & \textit{(Messages)} \end{array}$$

The types "$\mathsf{principal}$" and "nonce" classify principals and nonces, respectively. The next three productions allow distinguishing between shared keys, public keys and private keys. Dependent types offer a simple and flexible way to express the relations that hold between keys and their owner or other keys. A key "k" shared between principals "A" and "B" will have type "$\mathsf{shK\ A\ B}$". Here, the type of the key depends on the specific principals "A" and "B". Similarly, a constant "k" is given type "$\mathsf{pubK\ A}$" to indicate that it is a public key belonging to "A". We use dependent types again to express the relation between a public key and its inverse. Continuing with the last example, the inverse of "k" will have type "$\mathsf{privK\ k}$".

We use the type msg to classify generic messages. We reconcile nonces, keys, and principal identifiers with the messages they are routinely part of by imposing a subsorting relation between types, formalized by the judgment "$\tau :: \tau'$" (τ is a subsort of τ'). In this paper, each of the types discussed above, with the exception of private keys, is an subtype of msg:

$$\mathsf{principal} :: \mathsf{msg} \qquad \mathsf{nonce} :: \mathsf{msg} \qquad \mathsf{shK}\ A\ B :: \mathsf{msg} \qquad \mathsf{pubK}\ A :: \mathsf{msg}$$

We use dependent Cartesian products to assign a type to message tuples:

$$\begin{array}{llcll} \textit{Type tuples} & \vec{\tau} & ::= & \cdot & \textit{(Empty tuple)} \\ & & | & \Sigma x{:}\tau.\,\vec{\tau} & \textit{(Type tuple extension)} \end{array}$$

Whenever the variable x does not occur in $\vec{\tau}$, we will abbreviate $\Sigma x{:}\tau.\,\vec{\tau}$ as $\tau \times \vec{\tau}$.

We next give the syntax for MSR rules. At the core of a rule one can find a left-hand side and a right-hand side, described shortly. This nucleus is enclosed in a layer of universal quantifications that assign a type to every variable mentioned in the rule (with a few exceptions — see below):

$$\begin{array}{llcll} \textit{Rule:} & r & ::= & \textit{lhs} \rightarrow \textit{rhs} & \textit{(Rule core)} \\ & & | & \forall x : \tau.\, r & \textit{(Parameter closure)} \end{array}$$

The *left-hand side* or *antecedent* of a rule is a possibly empty multiset of predicates. In this paper, we will consider only two forms: the *network predicate*, $\mathsf{N}(t)$, models messages in transit in the network, while the *role state predicates* $L(\vec{e})$ is used to pass control (through the predicate symbol L) and data (by means of the elementary terms $\vec{e}$) from one rule to the next:

$$\begin{array}{llcll} \textit{Predicate sequences:} & \textit{lhs} & ::= & \cdot & \textit{(Empty predicate sequence)} \\ & & | & \textit{lhs},\ \mathsf{N}(t) & \textit{(Extension with a network predicate)} \\ & & | & \textit{lhs},\ L(\vec{e}) & \textit{(Extension with a role state predicate)} \end{array}$$

The *right-hand side* or *consequent* of a rule is a multiset of predicates as well, but it can be preceded by a sequence of existential declarations which mark data that should be generated freshly (typically nonces and short-term keys):

$$\begin{array}{llcll} \textit{Right-Hand sides:} & \textit{rhs} & ::= & \textit{lhs} & \textit{(Sequence of message predicates)} \\ & & | & \exists x : \tau.\, \textit{rhs} & \textit{(Fresh data generation)} \end{array}$$

In the past, crypto-protocols have often been presented as the temporal sequence of messages being transmitted during a "normal" run. Recent proposals champion a view that places the involved parties in the foreground. A protocol is then a collection of independent *roles* that communicate by exchanging messages, without any reference to runs of any kind. A role has an owner, the principal that executes it, and specifies the sequence of messages that he/she will send, possibly in response to receiving messages of some expected form. The actions undertaken by a principal executing a role are expressed as collection of rules, together with declarations for all the role state predicates they rely upon.

$$\begin{array}{llll} \textit{Rule collections:} & \rho & ::= & \cdot & \textit{(Empty role)} \\ & & | & \exists L : \vec{\tau}.\, \rho & \textit{(Role state predicate parameter declaration)} \\ & & | & r,\ \rho & \textit{(Extension with a rule)} \end{array}$$

The weak form of quantification over role state predicates prevents two executions of the same role to interfere with each other by using the same role state predicate name.

Finally, all the roles constituting a protocol are collected in a *protocol theory.* Roles come in two flavors: *generic roles* can be executed by any principal, while *anchored roles* are bound to one specific agent (typically a server or the intruder):

$$\begin{array}{llll} \textit{Protocol theories:} & \mathcal{P} & ::= & \cdot & \textit{(Empty protocol theory)} \\ & & | & \mathcal{P},\ \rho^{\forall A} & \textit{(Extension with a generic role)} \\ & & | & \mathcal{P},\ \rho^{\mathsf{A}} & \textit{(Extension with an anchored role)} \end{array}$$

6.1.2 Example

We will now make the above definitions concrete by applying them to the specification of an actual security protocol. The server-less variant of the Needham-Schroeder public-key protocol [NS78] is a two-party crypto-protocol aimed at authenticating the initiator A to the responder B (but not necessarily vice versa). It is expressed below as the expected run in the "usual notation".

$$\begin{array}{ll} 1. & A \to B\text{:}\ \ \{\!\{n_A\ A\}\!\}_{k_B} \\ 2. & B \to A\text{:}\ \ \{\!\{n_A\ n_B\}\!\}_{k_A} \\ 3. & A \to B\text{:}\ \ \{\!\{n_B\}\!\}_{k_B} \end{array}$$

In the first line, the initiator A encrypts a message consisting of a fresh piece of information, or nonce, n_A and her own identity with the public key k_B of the responder B, and sends it (ideally to B). The second line describes the action that B undertakes upon receiving and interpreting this message: he creates a nonce n_B of its own, combines it with A's nonce n_A, encrypts the outcome with A's public key k_A, and sends the resulting message out. Upon receiving this message in the third line, A accesses n_B and sends it back encrypted with k_B. The run is completed when B receives this message.

We will now express each role in turn in the syntax of MSR. For space reasons, we typeset homogeneous constituents, namely the universal variable declarations and the predicate sequences in the antecedent and consequent, in columns within each rule; we also rely on some minor abbreviation.

The initiator's actions are represented by the following two-rule role:

$$\left(\begin{array}{lcccl}
\multicolumn{5}{l}{\ \exists L : \Sigma B{:}\mathsf{principal}.\ \mathsf{pubK}\ B \times \mathsf{nonce}.} \\[4pt]
\begin{array}{l} \forall B : \mathsf{principal}. \\ \forall k_B : \mathsf{pubK}\ B. \end{array} & \cdot & \rightarrow & \exists n_A : \mathsf{nonce}. & \begin{array}{l} \mathsf{N}(\{\!\{ n_A\ A \}\!\}_{k_B}) \\ L(B, k_B, n_A) \end{array} \\[14pt]
\begin{array}{l} \forall B : \mathsf{principal}. \\ \forall k_B : \mathsf{pubK}\ B \\ \forall k_A : \mathsf{pubK}\ A. \\ \forall k'_A : \mathsf{privK}\ k_A. \\ \forall n_A, n_B : \mathsf{nonce}. \end{array} & \begin{array}{l} \mathsf{N}(\{\!\{ n_A\ n_B \}\!\}_{k_A}) \\ L(B, k_B, n_A) \end{array} & \rightarrow & & \mathsf{N}(\{\!\{ n_B \}\!\}_{k_B})
\end{array}\right)^{\forall A}$$

Clearly, any principal can engage in this protocol as an initiator (or a responder). Our encoding is therefore structured as a generic role. Let A be its postulated owner. The first rule formalizes the first line of the "usual notation" description of this protocol from A's point of view. It has an empty antecedent since initiation is unconditional in this protocol fragment. Its right-hand side uses an existential quantifier to mark the nonce n_A as fresh. The consequent contains the transmitted message and the role state predicate $L(B, k_B, n_A)$, necessary to enable the second rule of this protocol. The arguments of this predicate record variables used in the second rule.

The second rule encodes the last two lines of the "usual notation" description. It is applicable only if the initiator has executed the first rule (enforced by the presence of the role state predicate) and she receives a message of the appropriate form. Its consequent sends the last message of the protocol.

MSR provides a specific type for each variable appearing in these rules. The equivalent "usual notation" specification relies instead on natural language and conventions to convey this same information, with clear potential for ambiguity.

The responder is encoded as the generic role below, whose owner we have mnemonically called B. The first rule of this role collapses the two topmost lines of the "usual notation" specification of this protocol fragment from the receiver's point of view. The second rule captures the reception and successful interpretation of the last message in the protocol by B: this step is often overlooked. This rule has no consequent.

$$\left(\begin{array}{lcccl}
\multicolumn{5}{l}{\ \exists L : \mathsf{principal} \times \mathsf{nonce}.} \\[4pt]
\begin{array}{l} \forall k_B : \mathsf{pubK}\ B. \\ \forall k'_B : \mathsf{privK}\ k_B. \\ \forall A : \mathsf{principal}. \\ \forall k_A : \mathsf{pubK}\ A \\ \forall n_A : \mathsf{nonce}. \end{array} & \mathsf{N}(\{\!\{ n_A\ A \}\!\}_{k_B}) & \rightarrow & \exists n_B : \mathsf{nonce}. & \begin{array}{l} \mathsf{N}(\{\!\{ n_A\ n_B \}\!\}_{k_A}) \\ L(A, n_B) \end{array} \\[14pt]
\begin{array}{l} \forall k_B : \mathsf{pubK}\ B. \\ \forall k'_B : \mathsf{privK}\ k_B. \\ \forall A : \mathsf{principal}. \\ \forall n_B : \mathsf{nonce}. \end{array} & \begin{array}{l} \mathsf{N}(\{\!\{ n_B \}\!\}_{k_B}) \\ L(A, n_B) \end{array} & \rightarrow & & \cdot
\end{array}\right)^{\forall B}$$

6.1.3 Semantics

MSR supports two static checks and one dynamic behavior model, which altogether form the semantics of this formalism.

- The first static check is obviously type-checking, whose definition [Cer01a] is a simple adaptation of traditional schemes for dependently-typed languages. We will not display the typing rules of MSR in this paper, and we will implicitly encode their verification as type-checking for CLF terms.

- A more domain-specific test is *data access specification*, or *DAS*. It defines which data a given principal is allowed to access to construct or interpret messages. For example, it is admissible for an agent to look up the name and public key of any other principal, but it should be allowed to access only its own private key. Similarly, an agent cannot guess nonces, but is allowed to retrieve nonces it has memorized or received in a network message. We will completely ignore DAS in this paper, although it has very insightful properties and it will be very interesting to study them within CLF.

- Finally, MSR rules can be seen as a sophisticate form of Petri net transition: after instantiation, they rewrite the facts (tokens) in their antecedent (pre-conditions) to the facts (tokens) in their consequent (post-conditions). In this section, we will give a detailed account of this dynamic behavior.

At the core of the MSR equivalent of the Petri net notion of marking is a *state*. In line with the earlier interpretation of facts as overgrown tokens, a state is unsurprisingly a multiset of fully instantiated facts. We have the following grammatical productions:

$$\begin{array}{llcll}
\textit{States:} & S & ::= & \cdot & \textit{(Empty state)} \\
 & & | & S,\ \mathsf{N}(t) & \textit{(Extension with a network predicate)} \\
 & & | & S,\ \mathsf{L}(\vec{t}) & \textit{(Extension with a role state predicate)}
\end{array}$$

Observe that the role state predicates symbol themselves must be constants, while rules earlier required variable predicate names.

Because of the peculiarities of MSR, state do not provide a snapshot of execution in the same way as markings did in the case of Petri nets. Since the right-hand side of a rule can produce new symbols, it is important to keep an accurate account of what constants are in use at each instant of the execution. As usual, this is done by means of a signature:

$$\begin{array}{llcll}
\textit{Signatures} & \Sigma & ::= & \cdot & \textit{(Empty signature)} \\
 & & | & \Sigma,\ a : \tau & \textit{(Atomic message declaration)} \\
 & & | & \Sigma,\ \mathsf{L} : \vec{\tau} & \textit{(Local state predicate declaration)}
\end{array}$$

One more ingredient is needed: MSR rules are clustered into roles which specify the sequence of actions that a principal should perform, possibly in response to the reception of messages. Several instances of a given role, possibly stopped at different rules, can be present at any moment during execution. We record the role instances currently in use, the point at which each is stopped, and the principals who are executing them in an *active* *role set*. These objects are finite collections of *active roles*, *i.e.,* partially instantiated rule collections, each labelled with a principal name. The following grammar captures their macroscopic structure:

$$\begin{array}{llcll} \textit{Active role sets:} & R & ::= & \cdot & \textit{(Empty active role set)} \\ & & \mid & R,\ \rho^{\mathsf{A}} & \textit{(Extension with an instantiated role)} \end{array}$$

With the above definitions, we define a *configuration* as a triple $C = [S]_{\Sigma}^{R}$ consisting of a state S, a signature Σ, and an active role set R. Configurations are transformed by the application of MSR rules, and therefore correspond to the Petri net notion of marking.

Given a protocol $\mathcal{P}$, we describe the fact that execution transforms a configuration C into another configuration C' by means of the *one-step firing* judgment "$\mathcal{P} \triangleright C \longrightarrow C'$". It is implemented by the next six rules that fall into three classes. We should be able to: first, make a role from $\mathcal{P}$ available for execution; second, perform instantiations and apply a rule; and third, skip rules.

We first examine how to extend the current active role set R with a role taken from the protocol specification $\mathcal{P}$. As defined in Section 6.1.1, $\mathcal{P}$ can contain both anchored roles ρ^{A} and generic roles $\rho^{\forall A}$. This yields the following two rules, respectively:

$$\frac{}{(\mathcal{P}, \rho^{\mathsf{A}}) \triangleright [S]_{\Sigma}^{R} \longrightarrow [S]_{\Sigma}^{R,\rho^{\mathsf{A}}}}\ \textbf{ex\_arole} \qquad \frac{}{(\mathcal{P}, \rho^{\forall A}) \triangleright [S]_{\Sigma,\mathsf{A}:\mathsf{principal},\Sigma'}^{R} \longrightarrow [S]_{\Sigma,\mathsf{A}:\mathsf{principal},\Sigma'}^{R,([\mathsf{A}/A]\rho)^{\mathsf{A}}}}\ \textbf{ex\_grole}$$

Anchored roles can simply be copied to the current active role sets, while the owner of a generic role must be instantiated first to a principal. Here and below, we write $[o/x]o'$ for the capture-free substitution of object o for x in object o'.

Once a role has been activated, chances are that it contains role state predicate parameter declarations that require to be instantiated with actual constants before any of the embedded rules can be applied. This is done by the following rule where the newly generated constant L is added to the signature of the target configuration.

$$\frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(\exists L:\vec{\tau}.\,\rho)^{\mathsf{A}}} \longrightarrow [S]_{(\Sigma,\mathsf{L}:\vec{\tau})}^{R,([\mathsf{L}/L]\rho)^{\mathsf{A}}}}\ \textbf{ex\_rsp}$$

An exposed rule r can participate in an atomic execution step in two ways: we can either skip it (discussed below), or we can apply it to the current configuration. The latter option is implemented by the inference rule below, which makes use of the rule application judgment "$r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}$" (described shortly) to construct the state S' and the signature Σ' resulting from the application.

$$\frac{r \triangleright [S]_{\Sigma} \gg [S']_{\Sigma'}}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(r,\rho)^{\mathsf{A}}} \longrightarrow [S']_{\Sigma'}^{R,(\rho)^{\mathsf{A}}}}\ \textbf{ex\_rule}$$

The next two inference figures discard a rule (to implement non-deterministic behaviors) and remove active roles that have been completely executed, respectively.

$$\frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(r,\rho)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R,(\rho)^{\mathsf{A}}}}\ \textbf{ex\_skp} \qquad \frac{}{\mathcal{P} \triangleright [S]_{\Sigma}^{R,(\cdot)^{\mathsf{A}}} \longrightarrow [S]_{\Sigma}^{R}}\ \textbf{ex\_dot}$$

When successful, the application of a rule r to a state S in the signature Σ produces an updated state S' in the extended signature Σ'. This operation is defined by the *rule application* judgment "$r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}$". It is implemented by the following two rules, which respectively instantiate a universally quantified variable and perform the actual transition specified by the core of this rule:

$$\frac{\Sigma \vdash t : \tau \quad [t/x]r \triangleright [S]_\Sigma \gg [S']_{\Sigma'}}{(\forall x : \tau.\, r) \triangleright [S]_\Sigma \gg [S']_{\Sigma'}}\,\textbf{ex\_all} \qquad \frac{(\mathit{rhs})_\Sigma \gg (\mathit{lhs}')_{\Sigma'}}{(\mathit{lhs} \to \mathit{rhs}) \triangleright [S, \mathit{lhs}]_\Sigma \gg [S, \mathit{lhs}']_{\Sigma'}}\,\textbf{ex\_core}$$

Rule **ex_all** relies on the type-checking judgment $\Sigma \vdash t : \tau$ to generate a message of the appropriate type. Rule **ex_core** identifies the left-hand side *lhs* in the current state and replaces it with a substate lhs' derived from the consequent *rhs* by means of the right-hand side instantiation judgment "$(\mathit{rhs})_\Sigma \gg (\mathit{lhs}')_{\Sigma'}$" discussed next.

The *right-hand side instantiation* judgment instantiates every existentially quantified variable in a consequent *rhs* with a fresh constant of the appropriate type before returning the embedded predicate sequence:

$$\frac{([\mathsf{a}/x]\mathit{rhs})_{(\Sigma, \mathsf{a}:\tau)} \gg (\mathit{lhs})_{\Sigma'}}{(\exists x : \tau.\, \mathit{rhs})_\Sigma \gg (\mathit{lhs})_{\Sigma'}}\,\textbf{ex\_nnc} \qquad \frac{}{(\mathit{lhs})_\Sigma \gg (\mathit{lhs})_\Sigma}\,\textbf{ex\_seq}$$

The one-step firing judgment we just described corresponds to the application of a Petri net transition relative to a given marking. This operation can be used as the basis of either an interleaving semantics akin to what we described in Section 5.2.1, or of a trace semantics which extends the construction in 5.2.2. The former is defined, as expected, as the reflexive and transitive closure of the one-step execution judgment and will be denoted "$\mathcal{P} \triangleright C \longrightarrow^{(*)} C'$". The definition of the latter is a simple, but rather long extension of the analogous concept for Petri nets: we will not formalize it in this document. The interested reader can find formal definitions relative to an earlier version of MSR in [CDL$^+$00]. This same reference present a detailed analysis of the relation between these two form of semantics, and ultimately a proof of their equivalence.

6.2 CLF Encoding

We will now build on the encoding of Petri nets presented in Section 5. We describe the CLF representation of messages, rules and ultimately protocol theories in Section 6.2.1. We make these definition concrete in Section 6.2.2 by applying them to our on-going example. Finally, in Section 6.2.3, we state the adequacy theorems for this encoding, although we do not present proofs.

6.2.1 MSR

We will now describe the CLF representation of the various constituents of the MSR instance discussed in Section 6.1. We start with messages.

For the sake of simplicity, we will map MSR types directly to types in CLF. The grammar for the object types (left) yields the signature fragment displayed at right:

$$\left\lceil \begin{array}{rcl} \tau & ::= & \mathsf{principal} \\ & | & \mathsf{nonce} \\ & | & \mathsf{shK}\ A\ B \\ & | & \mathsf{pubK}\ A \\ & | & \mathsf{privK}\ k \\ & | & \mathsf{msg} \end{array} \right\rceil = \left\{ \begin{array}{ll} \mathsf{principal} & : \mathsf{type}. \\ \mathsf{nonce} & : \mathsf{type}. \\ \mathsf{shK} & : \mathsf{principal} \to \mathsf{principal} \to \mathsf{type}. \\ \mathsf{pubK} & : \mathsf{principal} \to \mathsf{type}. \\ \mathsf{privK} & : \Pi A\!:\!\mathsf{principal}.\ \mathsf{pubK}\ A \to \mathsf{type}. \\ \mathsf{msg} & : \mathsf{type}. \end{array} \right.$$

With this definition, type-checking in MSR will be emulated by the analogous check in CLF. A more detailed study of the type system of our object language (to prove type preservation results, or to talk about ill-typed messages, for example) would require a different encoding which represents MSR types as CLF objects.

The above definition does not reflect the presence of a subtyping relation in MSR. Since CLF does not natively support subtyping, we will emulate it by introducing a number of type coercion symbols that mediate between objects standing for principals, etc., and messages (of type msg). It should be noted that while this technical device is adequate for the simple instance of MSR considered in this document, it would not work in more general settings, in particular in situations where a sort has more than one supertype. A more complex encoding would then be required.

$$\left\lceil \begin{array}{ll} \mathsf{principal} & :: \mathsf{msg} \\ \mathsf{nonce} & :: \mathsf{msg} \\ \mathsf{shK}\ A\ B & :: \mathsf{msg} \\ \mathsf{pubK}\ A & :: \mathsf{msg} \end{array} \right\rceil = \left\{ \begin{array}{ll} \mathsf{p2m} & : \mathsf{principal} \to \mathsf{msg}. \\ \mathsf{n2m} & : \mathsf{nonce} \to \mathsf{msg}. \\ \mathsf{sk2m} & : \mathsf{shK}\ A\ B \to \mathsf{msg}. \\ \mathsf{pk2m} & : \mathsf{pubK}\ A \to \mathsf{msg}. \end{array} \right.$$

Here, the arguments A and B are implicitly Π-quantified at the head of the appropriate declarations. We assume that implicit arguments can be reconstructed, as is normally the case in LF.

Before displaying the encoding of messages, we need to introduce CLF constants that stand for their constructors. We will use the following declarations to represent concatenation, shared-key encryption (although not used in our example), and public-key encryption.

$$\begin{array}{lll} + & : \mathsf{msg} \to \mathsf{msg} \to \mathsf{msg}. & \textit{(infix)} \\ \mathsf{sEnc} & : \mathsf{shK}\ A\ B \to \mathsf{msg} \to \mathsf{msg}. & \\ \mathsf{pEnc} & : \mathsf{pubK}\ A \to \mathsf{msg} \to \mathsf{msg}. & \end{array}$$

With these definitions, we have the following encoding of messages:

$$\begin{array}{rlcl} \textit{(Principals)} & \ulcorner A \urcorner & = & \mathsf{p2m}\ A \\ \textit{(Nonce)} & \ulcorner n \urcorner & = & \mathsf{n2m}\ n \\ \textit{(Shared keys)} & \ulcorner k \urcorner & = & \mathsf{sk2m}\ k \\ \textit{(Public keys)} & \ulcorner k \urcorner & = & \mathsf{pk2m}\ k \\ \textit{(Concatenation)} & \ulcorner t_1\ t_2 \urcorner & = & \ulcorner t_1 \urcorner + \ulcorner t_2 \urcorner \\ \textit{(Symmetric-key encryption)} & \ulcorner \{t\}_k \urcorner & = & \mathsf{sEnc}\ k\ \ulcorner t \urcorner \\ \textit{(Asymmetric-key encryption)} & \ulcorner \{\!\{t\}\!\}_k \urcorner & = & \mathsf{pEnc}\ k\ \ulcorner t \urcorner \end{array}$$

Here, we assume we can tell shared keys from public keys, which is immediate with the help of a typing derivation.

We complete the presentation of the CLF constants needed for MSR with the following declarations for the network predicate and some infrastructure supporting role state predicates:

$$\begin{array}{ll} \mathsf{net} & : \mathsf{msg} \to \mathsf{type}. \\ \mathsf{rspArg} & : \mathsf{type}. \\ \mathsf{rsp} & : \mathsf{rspArg} \to \mathsf{type}. \end{array}$$

MSR facts are represented as CLF dependent types. For example, the network predicate $\mathsf{N}(\mathsf{n_B})$ is encoded as the type $\mathsf{net}\ (\mathsf{n2m}\ \mathsf{n_B})$. Role state predicates are however generated dynamically, but CLF does support not any form of quantification over type symbols. We resolve this issue by introducing the type rspArg: an MSR declaration $L{:}\vec{\tau}$ for a role-state predicate L will be represented by a CLF declaration for an object L that will take arguments as specified in τ, but whose target type will be rspArg. Thus, any occurrence of a fact $L(\vec{t})$ in a role will be encoded as the CLF type $\mathsf{rsp}\ (L\ulcorner\vec{t}\urcorner)$ as described below.

Let Σ_{msr} be the signature obtain by collecting all of the above declarations.

We now move to the representation of the higher syntactic level of an MSR specification. We start with rules and their component. The encoding $\ulcorner r\urcorner$ of a rule r is given as follows:

$$\begin{array}{lcl} \ulcorner \mathit{lhs} \to \mathit{rhs}\urcorner & = & \ulcorner \mathit{lhs}\urcorner^{\{\ulcorner \mathit{rhs}\urcorner\}} \\ \ulcorner \forall x:\tau.\, r\urcorner & = & \Pi x{:}\tau.\ulcorner r\urcorner \end{array}$$

The core of a rule is the object of the same representation technique seen in Section 5.3.1 for Petri net transitions. As formalized below, the antecedent will be rendered as a sequence of linear implications while the consequent is mapped to the monadic encapsulation of an asynchronous CLF formula. The outer layer of universal quantifiers is simply emulated by dependent types.

As anticipated, we unfold the left-hand side of a rule as a (possibly empty) sequence of linear implications:

$$\begin{array}{lcl} \ulcorner . \urcorner^{R} & = & R \\ \ulcorner \mathsf{N}(t),\ \mathit{lhs}\urcorner^{R} & = & \ulcorner \mathit{lhs}\urcorner^{(\mathsf{net}\,\ulcorner t\urcorner \multimap R)} \\ \ulcorner L(\vec{t}),\ \mathit{lhs}\urcorner^{R} & = & \ulcorner \mathit{lhs}\urcorner^{\mathsf{rsp}\,\ulcorner \vec{t}\urcorner^{L} \multimap R)} \end{array}$$

Here, R is a CLF expression representing the right-hand side of the rule. Role state predicates are encoded by applying each argument to the name of this predicate (we applied a similar technique in Section 5.3.3 to represent the application of a transition in an execution sequence) and feeding the result as an index to the type family rsp:

$$\begin{array}{lcl} \ulcorner . \urcorner^{M} & = & M \\ \ulcorner t, \vec{t}\urcorner^{N} & = & \ulcorner \vec{t}\urcorner^{M\,t} \end{array}$$

The facts in the right-hand side of a rule are tensored together while the existential quantifiers are mapped to the analogous constructs of CLF.

$$\begin{array}{lcl} \ulcorner \exists x:\tau.\, \mathit{rhs}\urcorner & = & \exists x:\tau.\, \ulcorner \mathit{rhs}\urcorner \\ \ulcorner . \urcorner & = & 1 \\ \ulcorner \mathsf{N}(t),\ \mathit{rhs}\urcorner & = & \mathsf{net}\,\ulcorner t\urcorner \otimes \mathit{rhs} \\ \ulcorner L(\vec{t}),\ \mathit{rhs}\urcorner & = & \mathsf{rsp}\,\ulcorner \vec{t}\urcorner^{L} \otimes \mathit{rhs} \end{array}$$

For the sake of brevity, we made this encoding slightly more general than necessary by allowing occurrences of the data generation construct embedded within a sequence of predicates. Clearly, this can easily be fixed.

A rule collection is encoded by tensoring together the representation of the individual rules that constitute it, while the role-state predicate declarations are mapped to existential constructions in CLF:

$$\begin{array}{lcl} \ulcorner . \urcorner & = & 1 \\ \ulcorner \exists L : \vec{\tau}.\, \rho \urcorner & = & \exists L : \ulcorner \vec{\tau} \urcorner .\, \ulcorner \rho \urcorner \\ \ulcorner r,\, \rho \urcorner & = & \ulcorner r \urcorner \otimes \ulcorner \rho \urcorner \end{array}$$

The type tuple $\vec{\tau}$ in this definition is curried as a iterated dependent types with rspArg as the target type:

$$\begin{array}{lcl} \ulcorner . \urcorner & = & \mathsf{rspArg} \\ \ulcorner \Sigma x{:}\tau.\, \vec{\tau} \urcorner & = & \Pi x : \tau.\, \ulcorner \vec{\tau} \urcorner \end{array}$$

Finally, a protocol theory is rendered in CLF by giving one declaration for each constituent role. Anchored role are represented by means of a monad containing the encoding of the corresponding rule collection. The representation of generic roles differs only by an additional quantification over its owner. In both cases, we assign a name to each role. This list of declarations is preceded by the common definitions for an MSR specification collected in Σ_{msr}.

$$\begin{array}{lcl} \ulcorner . \urcorner & = & \Sigma_{msr} \\ \ulcorner \mathcal{P},\, \rho^{\forall A} \urcorner & = & \ulcorner \mathcal{P} \urcorner,\ \mathsf{id}_\rho : \Pi A : \mathsf{principal}.\, \{ \ulcorner \rho \urcorner \} \\ \ulcorner \mathcal{P},\, \rho^{\mathsf{A}} \urcorner & = & \ulcorner \mathcal{P} \urcorner,\ \mathsf{id}_\rho : \{ \ulcorner \rho \urcorner \} \end{array}$$

6.2.2 Example

We now apply the representation function outlined in the previous section to our running example: the Needham-Schroeder public key protocol. The initiator role is represented by the following declaration:

$$\begin{array}{ll}
\mathsf{nspk_init} : & \Pi A : \mathsf{principal}. \\
& \{ \quad \exists L : \Pi B : \mathsf{principal}.\ \mathsf{pubK}\ B \rightarrow \mathsf{nonce} \rightarrow \mathsf{rspArg} \\
& \quad .\ \ \Pi B : \mathsf{principal}. \quad \Pi k_B : \mathsf{pubK}\ B. \\
& \qquad \{ \quad \exists n_A : \mathsf{nonce} \\
& \qquad\quad .\ \mathsf{net}\ (\mathsf{pEnc}\ k_B\ ((\mathsf{n2m}\ n_A) + (\mathsf{p2m}\ A))) \\
& \qquad \otimes\ \mathsf{rsp}\ (L\ B\ k_B\ n_A) \\
& \qquad \otimes\ 1 \ \} \\
& \ \ \otimes\ \ \Pi B : \mathsf{principal}. \quad \Pi k_B : \mathsf{pubK}\ B. \\
& \qquad \Pi k_A : \mathsf{pubK}\ A. \quad \Pi k'_A : \mathsf{privK}\ k_A. \\
& \qquad \Pi n_A : \mathsf{nonce}. \quad \Pi n_B : \mathsf{nonce}. \\
& \qquad\qquad \mathsf{net}\ (\mathsf{pEnc}\ k_A\ ((\mathsf{n2m}\ n_A) + (\mathsf{n2m}\ n_B))) \\
& \qquad\quad \multimap\ \mathsf{rsp}\ (L\ B\ k_B\ n_A) \\
& \qquad \multimap \{ \quad \mathsf{net}\ (\mathsf{pEnc}\ k_B\ (\mathsf{n2m}\ n_B)) \ \} \\
& \qquad\qquad \otimes\ 1 \ \} \\
& \ \ \otimes\ \ 1 \\
& \}
\end{array}$$

We invite the reader to compare this CLF declaration to the MSR role it represents (in Section 6.1.2). With the exception of a few details discussed in a moment, this translation is very direct. We have however the following discrepancies:

- MSR's subtyping is mapped to coercions in CLF. While this is acceptable in this example, this encoding would not be adequate for more sophisticated situations. Treating these cases would require either a more complex encoding, or the investigation of extensions to CLF.
- A role state predicate $L(\vec{t})$ is represented indirectly as the constant symbol rsp applied to a dynamically generated function symbol L that takes the representation of $\vec{t}$ as arguments. While CLF forces our hand on this, we believe that the definition of MSR could be seamlessly adapted to reflect a similar behavior.
- Rule consequents and roles are terminated by an omnipresent 1. We could have given a slightly more tedious encoding which removes unnecessary 1's.

The encoding of the responder is analogous and the object of similar remarks:

$$
\begin{array}{ll}
\mathsf{nspk\_resp}: & \Pi B{:}\mathsf{principal}.\\
& \{\quad \exists L : \mathsf{principal} \to \mathsf{principal} \to \mathsf{nonce} \to \mathsf{rspArg}\\
& \quad .\ \ \Pi k_B{:}\mathsf{pubK}\ B.\quad \Pi k'_B{:}\mathsf{privK}\ k_B.\\
& \qquad \Pi A{:}\mathsf{principal}.\quad \Pi k_A{:}\mathsf{pubK}\ A.\\
& \qquad \Pi n_A{:}\mathsf{nonce}.\\
& \qquad\qquad \mathsf{net}\ (\mathsf{pEnc}\ k_B\ ((\mathsf{n2m}\ n_A) + (\mathsf{p2m}\ A)))\\
& \qquad \multimap \{\quad \exists n_B : \mathsf{nonce}\\
& \qquad\qquad .\ \ \mathsf{net}\ (\mathsf{pEnc}\ k_A\ ((\mathsf{n2m}\ n_A) + (\mathsf{n2m}\ n_B)))\\
& \qquad\qquad \otimes\ \ \mathsf{rsp}\ (L\ A\ n_B)\\
& \qquad\qquad \otimes\ \ 1\ \}\\
& \quad \otimes\ \ \Pi A{:}\mathsf{principal}.\\
& \qquad \Pi k_B{:}\mathsf{pubK}\ B.\quad \Pi k'_B{:}\mathsf{privK}\ k_B.\\
& \qquad \Pi n_B{:}\mathsf{nonce}.\\
& \qquad\qquad \mathsf{net}\ (\mathsf{pEnc}\ k_B\ (\mathsf{n2m}\ n_B))\\
& \qquad \multimap\ \ \mathsf{rsp}\ (L\ A\ n_B)\\
& \qquad \multimap\ \ \{\ 1\ \}\\
& \quad \otimes\ \ 1\\
& \}
\end{array}
$$

6.2.3 Adequacy Results

In order to discuss the adequacy of the representation of MSR just presented, we need to define the representation of the execution judgment $\mathcal{P} \triangleright C \ \longrightarrow^{(*)}\ C'$ of this language. Similarly to the case of Petri net markings, the two sides of this judgment will be have distinct representation: the antecedent C will be encoded as a context, while the consequent C' as a synchronous CLF formula. The protocol theory $\mathcal{P}$ becomes the signature of the corresponding CLF judgment.

We start by giving the representation of configuration on the left-hand side of the multi-step execution judgment. We map the signature part to a sequence of declarations in the unrestricted CLF context, while the state and the active role components are translated as a set of linear declarations:

$$\ulcorner [S]^{R}_{\Sigma} \urcorner^{(1)} = \underbrace{\ulcorner \Sigma \urcorner^{(1)}}_{\Gamma} ; \underbrace{(\ulcorner R \urcorner^{(1)}, \ulcorner S \urcorner^{(1)})}_{\Delta}$$

The encoding of a signature on the left-hand side simply discharges its constituents as unrestricted context declarations:

$$\begin{array}{lcl} \ulcorner . \urcorner^{(1)} & = & \cdot \\ \ulcorner \Sigma, a : \tau \urcorner^{(1)} & = & \ulcorner \Sigma \urcorner^{(1)}, a : \tau \\ \ulcorner \Sigma, \mathsf{L} : \vec{\tau} \urcorner^{(1)} & = & \ulcorner \Sigma \urcorner^{(1)}, \mathsf{L} : \ulcorner \vec{\tau} \urcorner \end{array}$$

Recall that MSR types are mapped to CLF types. The encoding of type tuples $\ulcorner \vec{\tau} \urcorner$ was given earlier.

Each element in a state S is rendered as a linear declaration in CLF. Clearly, we need to prefix each declaration with a distinct variable name (which we generically call x here). The encoding of the individual elements was given in Section 6.2.1.

$$\begin{array}{lcl} \ulcorner . \urcorner^{(1)} & = & \cdot \\ \ulcorner S, \mathsf{N}(t) \urcorner^{(1)} & = & \ulcorner S \urcorner^{(1)},\ x \hat{:} \mathsf{net}\, \ulcorner t \urcorner \\ \ulcorner S, \mathsf{L}_l(\vec{t}) \urcorner^{(1)} & = & \ulcorner S \urcorner^{(1)},\ x \hat{:} \mathsf{rsp}\, \ulcorner \vec{t} \urcorner^{L} \end{array}$$

The left-hand encoding of active roles is similar:

$$\begin{array}{lcl} \ulcorner . \urcorner^{(1)} & = & \cdot \\ \ulcorner R, \rho^{\mathsf{A}} \urcorner^{(1)} & = & \ulcorner R \urcorner^{(1)},\ z \hat{:} \ulcorner \rho \urcorner \end{array}$$

We now turn to configurations $C' = [S']^{R'}_{\Sigma'}$ that appear on the right-hand side of the MSR execution judgment. Intuitively, we want to represent C as a synchronous CLF formula obtained by tensoring the encodings of the components of the state S' and the active role set R', prefixed by existential quantifications over the signature Σ'. This is however inadequate unless C' is reached from a configuration with an empty signature. In general, we must omit a prefix Σ of the signature, corresponding to the constants available at the beginning of the considered execution sequence. The encoding of C' is therefore relative to Σ. This intuition is formalized as follows, where Σ' has been expanded as Σ, Σ'':

$$\ulcorner [S']^{R'}_{\Sigma,\Sigma''} \urcorner^{(2'')}_{|\Sigma} = \ulcorner \Sigma'' \urcorner^{(2'')}_{(\ulcorner R' \urcorner^{(2'')} \otimes \ulcorner S' \urcorner^{(2'')})}$$

We anticipated that the encoding of the state and active role set components of a configuration is obtained by tensoring the representation of their constituents. This is formalized as follows:

$$\begin{array}{lcl} \ulcorner . \urcorner^{(2'')} & = & 1 \\ \ulcorner S, \mathsf{N}(t) \urcorner^{(2'')} & = & \ulcorner S \urcorner^{(2'')} \otimes \mathsf{net}\, \ulcorner t \urcorner \\ \ulcorner S, \mathsf{L}_l(\vec{t}) \urcorner^{(2'')} & = & \ulcorner S \urcorner^{(2'')} \otimes \mathsf{rsp}\, \ulcorner \vec{t} \urcorner^{L} \end{array} \qquad \begin{array}{lcl} \ulcorner . \urcorner^{(2'')} & = & 1 \\ \ulcorner R, \rho^{\mathsf{A}} \urcorner^{(2'')} & = & \ulcorner R \urcorner^{(2'')} \otimes \ulcorner \rho \urcorner \end{array}$$

The result of this operation is prefixed by a sequence of existential quantifiers guided by the signature fragment under examination:

$$
\begin{array}{lcl}
\ulcorner \cdot \urcorner^{(2'')}_{S} & = & S \\
\ulcorner \Sigma', a : \tau \urcorner^{(2'')}_{S} & = & \ulcorner \Sigma' \urcorner^{(2'')}_{\exists a:\tau.\, S} \\
\ulcorner \Sigma', \mathsf{L}{:}\vec{\tau} \urcorner^{(2'')}_{S} & = & \ulcorner \Sigma' \urcorner^{(2'')}_{\exists \mathsf{L}:\ulcorner \vec{\tau} \urcorner.\, S}
\end{array}
$$

With the help of these definition, we propose the following statement for the adequacy of our representation of MSR. While we have not formally proved it, we expect that a proof can be achieved by using the proof of adequacy for Petri nets given in Section 5.3.4 as a mold, and refining it to accommodate the idiosyncrasies of MSR.

Expected Result 6.1. (*Adequacy of the representation of MSR execution in CLF*)
Let $\mathcal{P}$ be a protocol theory and C and C' two configurations with $C = [S]^{R}_{\Sigma}$. There is a bijection between derivations of the MSR multi-step execution judgment

$$\mathcal{P} \rhd C \longrightarrow^{(*)} C'$$

and terms E such that the judgment

$$\ulcorner C \urcorner^{(1)} \vdash_{\ulcorner \mathcal{P} \urcorner} E \leftarrow \ulcorner C' \urcorner^{(2'')}_{|\Sigma}$$

is derivable in CLF.

The structure of the term E mentioned in this result is similar to what we obtained in the case of Petri nets. Rule applications are mapped to let expressions exactly like transition in Section 5.3.4. The presence of existential quantifiers account however for a richer encoding. Finally, the manipulation of roles is another source of let terms.

7 Conclusions

CLF extends the expressive power of the LF family of logical frameworks by permitting natural and adequate encodings of a large class of concurrent systems. This technical report evaluates this new formalism on four case studies which all embed different models of concurrency: the π-calculus, Concurrent ML with futures à la Multilisp, Petri nets, and the multiset rewriting based security protocol specification language MSR. One of the next tests for the framework will be the development of techniques that allow us to state and prove properties of computations in CLF and to explore formal encodings of the metatheory of our applications.

Acknowledgments

We would like to thank Mike Mislove for discussions on the nature of true concurrency.

A Syntax and judgments of CLF

A.1 Syntax

Definition 11 (Type constructors).

$$\begin{array}{rll} A, B, C & ::= A \multimap B \mid \Pi x{:}A.\, B \mid A \,\&\, B \mid \top \mid \{S\} \mid P & \textit{Asynchronous types} \\ P & ::= a \mid P\ N & \textit{Atomic type constructors} \\ S & ::= S_1 \otimes S_2 \mid 1 \mid \exists x{:}A.\, S \mid A & \textit{Synchronous types} \end{array}$$

Definition 12 (Kinds).

$$K, L ::= \mathsf{type} \mid \Pi x{:}A.\, K \qquad \textit{Kinds}$$

Definition 13 (Objects).

$$\begin{array}{rll} N & ::= \hat{\lambda} x.\, N \mid \lambda x.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R & \textit{Normal objects} \\ R & ::= c \mid x \mid R \,\hat{}\, N \mid R\ N \mid \pi_1 R \mid \pi_2 R & \textit{Atomic objects} \\ E & ::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M & \textit{Expressions} \\ M & ::= M_1 \otimes M_2 \mid 1 \mid [N, M] \mid N & \textit{Monadic objects} \\ p & ::= p_1 \otimes p_2 \mid 1 \mid [x, p] \mid x & \textit{Patterns} \end{array}$$

A.2 Equality

Definition 14 (Concurrent contexts).

$$\epsilon ::= \_ \mid \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \epsilon \qquad \textit{Concurrent contexts}$$

Definition 15 (Equality).

$E_1 =_{\mathrm{c}} E_2$ *[Concurrent equality]*

$$\frac{M_1 = M_2}{M_1 =_{\mathrm{c}} M_2} \qquad \frac{R_1 = R_2 \quad E_1 =_{\mathrm{c}} \epsilon[E_2]}{(\mathsf{let}\ \{p\} = R_1\ \mathsf{in}\ E_1) =_{\mathrm{c}} \epsilon[\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2]}\ *$$

$E_1 = E_2$ *[Expression equality]*

$$\frac{E_1 =_{\mathrm{c}} E_2}{E_1 = E_2}$$

$N_1 = N_2 \quad R_1 = R_2 \quad M_1 = M_2 \quad P_1 = P_2$ *[Other equalities]*

(All congruences.)

The rule marked (*) is subject to the side condition that no variable bound by p be free in the conclusion or bound by the context ϵ, and that no variable free in R_2 be bound by the context ϵ.

A.3 Instantiation

Definition 16 (Instantiation).

$\mathsf{treduce}_A(x.\, R) \equiv B$ *[Type reduction]*

$$
\begin{aligned}
&\mathsf{treduce}_A(x.\, x) \equiv A\\
&\mathsf{treduce}_A(x.\, R\ N) \equiv C \quad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv \Pi y\!:\!B.\, C\\
&\mathsf{treduce}_A(x.\, R^\wedge N) \equiv C \quad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv B \multimap C\\
&\mathsf{treduce}_A(x.\, \pi_1 R) \equiv B_1 \quad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv B_1 \,\&\, B_2\\
&\mathsf{treduce}_A(x.\, \pi_2 R) \equiv B_2 \quad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv B_1 \,\&\, B_2
\end{aligned}
$$

$\mathsf{reduce}_A(x.\, R, N_0) \equiv N'$ *[Reduction]*

$$
\begin{aligned}
&\mathsf{reduce}_A(x.\, x, N_0) \equiv N_0\\
&\mathsf{reduce}_A(x.\, R\ N, N_0) \equiv \mathsf{inst\_n}_B(y.\, N', \mathsf{inst\_n}_A(x.\, N, N_0))\\
&\qquad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv \Pi y\!:\!B.\, C \ \textit{and}\ \mathsf{reduce}_A(x.\, R, N_0) \equiv \lambda y.\, N'\\
&\mathsf{reduce}_A(x.\, R^\wedge N, N_0) \equiv \mathsf{inst\_n}_B(y.\, N', \mathsf{inst\_n}_A(x.\, N, N_0))\\
&\qquad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv B \multimap C \ \textit{and}\ \mathsf{reduce}_A(x.\, R, N_0) \equiv \hat{\lambda} y.\, N'\\
&\mathsf{reduce}_A(x.\, \pi_1 R, N_0) \equiv N_1' \quad \textit{if}\ \mathsf{reduce}_A(x.\, R, N_0) \equiv \langle N_1', N_2' \rangle\\
&\mathsf{reduce}_A(x.\, \pi_2 R, N_0) \equiv N_2' \quad \textit{if}\ \mathsf{reduce}_A(x.\, R, N_0) \equiv \langle N_1', N_2' \rangle
\end{aligned}
$$

$\mathsf{inst\_r}_A(x.\, R, N_0) \equiv R'$ *[Atomic object instantiation]*

$$
\begin{aligned}
&\mathsf{inst\_r}_A(x.\, c, N_0) \equiv c\\
&\mathsf{inst\_r}_A(x.\, y, N_0) \equiv y \quad \textit{if}\ y\ \textit{is not}\ x\\
&\mathsf{inst\_r}_A(x.\, R\ N, N_0) \equiv (\mathsf{inst\_r}_A(x.\, R, N_0))\ (\mathsf{inst\_n}_A(x.\, N, N_0))\\
&\mathsf{inst\_r}_A(x.\, R^\wedge N, N_0) \equiv (\mathsf{inst\_r}_A(x.\, R, N_0))^\wedge(\mathsf{inst\_r}_A(x.\, N, N_0))\\
&\mathsf{inst\_r}_A(x.\, \pi_1 R, N_0) \equiv \pi_1(\mathsf{inst\_r}_A(x.\, R, N_0))\\
&\mathsf{inst\_r}_A(x.\, \pi_2 R, N_0) \equiv \pi_2(\mathsf{inst\_r}_A(x.\, R, N_0))
\end{aligned}
$$

$\mathsf{inst\_n}_A(x.\, N, N_0) \equiv N'$ *[Normal object instantiation]*

$$
\begin{aligned}
&\mathsf{inst\_n}_A(x.\, \lambda y.\, N, N_0) \equiv \lambda y.\, \mathsf{inst\_n}_A(x.\, N, N_0) \quad \textit{if}\ y \notin \mathrm{FV}(N_0)\\
&\mathsf{inst\_n}_A(x.\, \hat{\lambda} y.\, N, N_0) \equiv \hat{\lambda} y.\, \mathsf{inst\_n}_A(x.\, N, N_0) \quad \textit{if}\ y \notin \mathrm{FV}(N_0)\\
&\mathsf{inst\_n}_A(x.\, \langle N_1, N_2 \rangle, N_0) \equiv \langle \mathsf{inst\_n}_A(x.\, N_1, N_0), \mathsf{inst\_n}_A(x.\, N_2, N_0) \rangle\\
&\mathsf{inst\_n}_A(x.\, \langle\rangle, N_0) \equiv \langle\rangle\\
&\mathsf{inst\_n}_A(x.\, \{E\}, N_0) \equiv \{\mathsf{inst\_e}_A(x.\, E, N_0)\}\\
&\mathsf{inst\_n}_A(x.\, R, N_0) \equiv \mathsf{inst\_r}_A(x.\, R, N_0) \quad \textit{if}\ \mathrm{head}(R)\ \textit{is not}\ x\\
&\mathsf{inst\_n}_A(x.\, R, N_0) \equiv \mathsf{reduce}_A(x.\, R, N_0) \quad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv P
\end{aligned}
$$

$\mathsf{inst\_m}_A(x.\, M, N_0) \equiv M'$ *[Monadic object instantiation]*

$$
\begin{aligned}
&\mathsf{inst\_m}_A(x.\, M_1 \otimes M_2, N_0) \equiv \mathsf{inst\_m}_A(x.\, M_1, N_0) \otimes \mathsf{inst\_m}_A(x.\, M_2, N_0)\\
&\mathsf{inst\_m}_A(x.\, 1, N_0) \equiv 1\\
&\mathsf{inst\_m}_A(x.\, [N, M], N_0) \equiv [\mathsf{inst\_n}_A(x.\, N, N_0), \mathsf{inst\_m}_A(x.\, M, N_0)]\\
&\mathsf{inst\_m}_A(x.\, N, N_0) \equiv \mathsf{inst\_n}_A(x.\, N, N_0)
\end{aligned}
$$

$\mathsf{inst\_e}_A(x.\, E, N_0) \equiv E'$ *[Expression instantiation]*

$$
\begin{aligned}
&\mathsf{inst\_e}_A(x.\, \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv (\mathsf{let}\ \{p\} = \mathsf{inst\_r}_A(x.\, R, N_0)\ \mathsf{in}\ \mathsf{inst\_e}_A(x.\, E, N_0))\\
&\qquad \textit{if}\ \mathrm{head}(R)\ \textit{is not}\ x,\\
&\qquad \textit{and}\ \mathrm{FV}(p) \cap \mathrm{FV}(N_0)\ \textit{is empty}\\
&\mathsf{inst\_e}_A(x.\, \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv \mathsf{match\_e}_S(p.\, \mathsf{inst\_e}_A(x.\, E, N_0), E')\\
&\qquad \textit{if}\ \mathsf{treduce}_A(x.\, R) \equiv \{S\},\ \mathsf{reduce}_A(x.\, R, N_0) \equiv \{E'\},\\
&\qquad \textit{and}\ \mathrm{FV}(p) \cap \mathrm{FV}(N_0)\ \textit{is empty}\\
&\mathsf{inst\_e}_A(x.\, M, N_0) \equiv \mathsf{inst\_m}_A(x.\, M, N_0)
\end{aligned}
$$

$\mathsf{match\_m}_S(p.\, E, M_0) \equiv E'$ *[Match monadic object]*

$$
\begin{aligned}
&\mathsf{match\_m}_{S_1 \otimes S_2}(p_1 \otimes p_2.\, E, M_1 \otimes M_2) \equiv \mathsf{match\_m}_{S_2}(p_2.\, \mathsf{match\_m}_{S_1}(p_1.\, E, M_1), M_2)\\
&\qquad \textit{if}\ \mathrm{FV}(p_2) \cap \mathrm{FV}(M_1)\ \textit{is empty}\\
&\mathsf{match\_m}_1(1.\, E, 1) \equiv E\\
&\mathsf{match\_m}_{\exists x : A. S}([x, p].\, E, [N, M]) \equiv \mathsf{match\_m}_S(p.\, \mathsf{inst\_e}_A(x.\, E, N), M)\\
&\qquad \textit{if}\ \mathrm{FV}(p) \cap \mathrm{FV}(N)\ \textit{is empty}\\
&\mathsf{match\_m}_A(x.\, E, N) \equiv \mathsf{inst\_e}_A(x.\, E, N)
\end{aligned}
$$

$\mathsf{match\_e}_S(p.\, E, E_0) \equiv E'$ *[Match expression]*

$$
\begin{aligned}
&\mathsf{match\_e}_S(p.\, E, \mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ E_0) \equiv \mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ \mathsf{match\_e}_S(p.\, E, E_0)\\
&\qquad \textit{if}\ \mathrm{FV}(p_0) \cap \mathrm{FV}(E)\ \textit{and}\ \mathrm{FV}(p) \cap \mathrm{FV}(E_0)\ \textit{are empty}\\
&\mathsf{match\_e}_S(p.\, E, M_0) \equiv \mathsf{match\_m}_S(p.\, E, M_0)
\end{aligned}
$$

$\mathsf{inst\_p}_A(x.\, P, N_0) \equiv P'$ *[Atomic type constructor instantiation]*
$\mathsf{inst\_a}_A(x.\, A, N_0) \equiv A'$ *[Type instantiation]*
$\mathsf{inst\_s}_A(x.\, S, N_0) \equiv S'$ *[Synchronous type instantiation]*
$\mathsf{inst\_k}_A(x.\, K, N_0) \equiv K'$ *[Kind instantiation]*

(Analogous.)

A.4 Expansion

Definition 17 (Expansion).

$\mathsf{expand}_A(R) \equiv N$ *[Expansion]*

$$\mathsf{expand}_P(R) \equiv R$$
$$\mathsf{expand}_{A \multimap B}(R) \equiv \hat{\lambda} x.\, \mathsf{expand}_B(R\,\hat{}\,(\mathsf{expand}_A(x))) \quad \textit{if } x \notin \mathrm{FV}(R)$$
$$\mathsf{expand}_{\Pi x:A.B}(R) \equiv \lambda x.\, \mathsf{expand}_B(R\ (\mathsf{expand}_A(x))) \quad \textit{if } x \notin \mathrm{FV}(R)$$
$$\mathsf{expand}_{A \& B}(R) \equiv \langle \mathsf{expand}_A(\pi_1 R), \mathsf{expand}_B(\pi_2 R) \rangle$$
$$\mathsf{expand}_{\top}(R) \equiv \langle\rangle$$
$$\mathsf{expand}_{\{S\}}(R) \equiv (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \mathsf{pexpand}_S(p))$$

$\mathsf{pexpand}_S(p) \equiv M$ *[Pattern expansion]*

$$\mathsf{pexpand}_{S_1 \otimes S_2}(p_1 \otimes p_2) \equiv \mathsf{pexpand}_{S_1}(p_1) \otimes \mathsf{pexpand}_{S_2}(p_2)$$
$$\mathsf{pexpand}_{1}(1) \equiv 1$$
$$\mathsf{pexpand}_{\exists x:A.S}([x, p]) \equiv [\mathsf{expand}_A(x), \mathsf{pexpand}_S(p)]$$
$$\mathsf{pexpand}_A(x) \equiv \mathsf{expand}_A(x)$$

A.5 Typing

Definition 18 (Signatures and contexts).

$\Sigma ::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$	*Signatures*
$\Gamma ::= \cdot \mid \Gamma, x{:}A$	*Unrestricted contexts*
$\Delta ::= \cdot \mid \Delta, x \hat{:} A$	*Linear contexts*
$\Psi ::= \cdot \mid p \hat{:} S, \Psi$	*Pattern contexts*

Definition 19 (Typing).

$\vdash \Sigma\ \mathsf{ok}$ *[Signature validity]*

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

$\vdash_\Sigma \Gamma\ \mathsf{ok}$ *[Context validity]*

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

$\Gamma \vdash_\Sigma \Delta\ \mathsf{ok}$ *[Linear context validity]*

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

$\Gamma \vdash_\Sigma \Psi\ \mathsf{ok}$ *[Pattern context validity]*

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

$\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind}$ *[Kind checking]*

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

$\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}$ *[Type checking]*

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

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash A \multimap B \Leftarrow \mathsf{type}}\ {\multimap}\mathbf{F}$$

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

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

$\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type}$ *[Synchronous type checking]*

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

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

$\Gamma \vdash_\Sigma P \Rightarrow K$ *[Atomic type constructor inference]*

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

$\Gamma \vdash_\Sigma N \Leftarrow A$ *[Normal object checking]*

$$\frac{\Gamma, x{:}A; \Delta \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \lambda x.\, N \Leftarrow \Pi x{:}A.\, B}\ \Pi\mathbf{I} \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow P' \quad P' = P}{\Gamma; \Delta \vdash R \Leftarrow P}\ {\Rightarrow}{\Leftarrow}$$

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

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

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta \vdash \{E\} \Leftarrow \{S\}}\ \{\}\mathbf{I}$$

$\Gamma \vdash_\Sigma R \Rightarrow A$ *[Atomic object inference]*

$$\frac{}{\Gamma; \cdot \vdash c \Rightarrow \Sigma(c)}\ c \qquad \frac{}{\Gamma; \cdot \vdash x \Rightarrow \Gamma(x)}\ x \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow \Pi x{:}A.\, B \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \Delta \vdash R\ N \Rightarrow \mathsf{inst\_a}_A(x.\, B, N)}\ \Pi\mathbf{E}$$

$$\frac{}{\Gamma; x \hat{:} A \vdash x \Rightarrow A}\ x \qquad \frac{\Gamma; \Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma; \Delta_2 \vdash N \Leftarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash R\,{}^\wedge N \Rightarrow B}\ {\multimap}\mathbf{E}$$

$$\frac{\Gamma; \Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma; \Delta \vdash \pi_1 R \Rightarrow A}\ \&\mathbf{E}_1 \qquad \frac{\Gamma; \Delta \vdash R \Rightarrow A \,\&\, B}{\Gamma; \Delta \vdash \pi_2 R \Rightarrow B}\ \&\mathbf{E}_2$$

$\Gamma; \Delta \vdash_\Sigma E \leftarrow S$ *[Expression checking]*

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

$\Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S$ *[Pattern expansion]*

$$\frac{\Gamma; \Delta \vdash E \leftarrow S}{\Gamma; \Delta; \cdot \vdash E \leftarrow S}\ \leftarrow\leftarrow$$

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

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

$\Gamma; \Delta \vdash_\Sigma M \Leftarrow S$ *[Monadic object checking]*

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

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

B Adequacy of the Synchronous π-calculus

For the sake of brevity, we combine unrestricted (Γ) and linear (Δ) contexts into a single context Γ throughout this appendix. We also omit leading π-quantifiers and the corresponding applications.

B.1 Syntax and Semantics

B.1.1 Syntax

Process Expressions	$P ::= (P \mid Q) \mid \mathsf{new}\, u\, P \mid\ !P \mid M$
Sums	$M ::= \mathsf{0} \mid c + M$
Actions	$c ::= \tau.P \mid u(v).P \mid \bar{u}{<}v{>}.P$

B.1.2 Structural Equivalence

$$\frac{}{P \equiv P \mid 0}\ str_1 \qquad \frac{}{P \mid Q \equiv Q \mid P}\ str_2 \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}\ str_3 \qquad \frac{}{0 \equiv \mathsf{new}\, u\, 0}\ str_4$$

$$\frac{}{\mathsf{new}\, u\, (P \mid Q) \equiv P \mid (\mathsf{new}\, u\, Q)}\ str_5\ \text{(if } u \notin P\text{)} \qquad \frac{}{\mathsf{new}\, u\, \mathsf{new}\, v\, P \equiv \mathsf{new}\, v\, \mathsf{new}\, u\, P}\ str_6$$

$$\frac{}{P \equiv P}\ str_7 \qquad \frac{P \equiv Q}{Q \equiv P}\ str_8 \qquad \frac{P \equiv R \quad R \equiv Q}{P \equiv Q}\ str_9$$

$$\frac{P \equiv P'}{P \mid Q \equiv P' \mid Q}\ str_{10} \qquad \frac{P \equiv P'}{\mathsf{new}\, u\, P \equiv \mathsf{new}\, u\, P'}\ str_{11} \qquad \frac{}{!P \equiv !P \mid P}\ str_{12} \qquad \frac{P \equiv P'}{!P \equiv !P'}\ str_{13}$$

$$\frac{M \equiv M'}{c + M \equiv c + M'}\ str_{14} \qquad \frac{c \equiv c'}{c + M \equiv c' + M}\ str_{15} \qquad \frac{}{c_1 + c_2 + M \equiv c_2 + c_1 + M}\ str_{16}$$

$$\frac{P \equiv P'}{\tau.P \equiv \tau.P'}\ str_{17} \qquad \frac{P \equiv P'}{u(v).P \equiv u(v).P'}\ str_{18} \qquad \frac{P \equiv P'}{\overline{u}\langle v\rangle.P \equiv \overline{u}\langle v\rangle.P'}\ str_{19}$$

B.1.3 Reduction

$$\frac{}{\tau.P + M \longrightarrow P}\ red_1 \qquad \frac{}{(u(y).P + M_1) \mid (\overline{u}\langle v\rangle.Q + M_2) \longrightarrow P[v/y] \mid Q}\ red_2$$

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}\ red_3 \qquad \frac{P \longrightarrow P'}{\mathsf{new}\, u\, P \longrightarrow \mathsf{new}\, u\, P'}\ red_4$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ red_5$$

B.1.4 Multi-Step Reduction

$$\frac{P \equiv Q}{P \longrightarrow^* Q}\ red_6 \qquad \frac{P \longrightarrow^* R \quad R \longrightarrow Q}{P \longrightarrow^* Q}\ red_7$$

B.1.5 Normal Processes

To simplify our adequacy proof, we borrow the idea of a process in standard form (which we call a normal process) from Milner.

Definition 20. (*Normal Process F*)
$F ::= \mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k$

Definition 21. (*Normal Reduction $\rightarrow^-$*)
Normal reductions have either of the following forms:

- $\mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid \tau.P$

  $\rightarrow^-$

  $\mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid P$

- $\mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid u(w).P + M_1' \mid \overline{u}\langle v\rangle.Q + M_2'$
  $\rightarrow^-$
  $\mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid P[v/w] \mid Q$

Lemma 22.
If $P \rightarrow Q$, then there exists an F such that $P \equiv F$ and $F \rightarrow^- Q'$ and $Q' \equiv Q$.

Proof. By induction on the derivation $P \rightarrow Q$. ☐

B.2 The Encoding

B.2.1 Syntactic Classes

$$
\begin{array}{lcl}
\mathsf{chan} & : & \mathsf{type}. \\
\mathsf{expr} & : & \mathsf{type}. \\
\mathsf{sum} & : & \mathsf{type}. \\
\mathsf{act} & : & \mathsf{type}.
\end{array}
$$

B.2.2 Pi-Calculus Terms

$$
\begin{array}{lcl}
\mathsf{par} & : & \mathsf{expr} \rightarrow \mathsf{expr} \rightarrow \mathsf{expr}. \\
\mathsf{new} & : & (\mathsf{chan} \rightarrow \mathsf{expr}) \rightarrow \mathsf{expr}. \\
\mathsf{rep} & : & \mathsf{expr} \rightarrow \mathsf{expr}. \\
\mathsf{sync} & : & \mathsf{sum} \rightarrow \mathsf{expr}. \\
\mathsf{null} & : & \mathsf{sum}. \\
\\
\mathsf{alt} & : & \mathsf{act} \rightarrow \mathsf{sum} \rightarrow \mathsf{sum}. \\
\mathsf{silent} & : & \mathsf{expr} \rightarrow \mathsf{act}. \\
\\
\mathsf{in} & : & \mathsf{chan} \rightarrow (\mathsf{chan} \rightarrow \mathsf{expr}) \rightarrow \mathsf{act}. \\
\mathsf{out} & : & \mathsf{chan} \rightarrow \mathsf{chan} \rightarrow \mathsf{expr} \rightarrow \mathsf{act}. \\
\\
\mathsf{proc} & : & \mathsf{expr} \rightarrow \mathsf{type}.
\end{array}
$$

B.2.3 Representation of syntax

$$
\begin{aligned}
\ulcorner P \mid Q \urcorner &= \mathsf{par}\, \ulcorner P \urcorner\, \ulcorner Q \urcorner \\
\ulcorner \mathsf{new}\, u\, P \urcorner &= \mathsf{new}\ (\lambda u. \ulcorner P \urcorner) \\
\ulcorner !P \urcorner &= \mathsf{rep}\, \ulcorner P \urcorner \\
\ulcorner M \urcorner &= \mathsf{sync}\, \ulcorner\!\ulcorner M \urcorner\!\urcorner
\end{aligned}
$$

$$
\begin{aligned}
\ulcorner\!\ulcorner 0 \urcorner\!\urcorner &= \mathsf{null} \\
\ulcorner\!\ulcorner c + M \urcorner\!\urcorner &= \mathsf{alt}\, \ulcorner\!\ulcorner\!\ulcorner c \urcorner\!\urcorner\!\urcorner\, \ulcorner\!\ulcorner M \urcorner\!\urcorner
\end{aligned}
$$

$$
\begin{aligned}
\ulcorner\!\ulcorner\!\ulcorner \tau.P \urcorner\!\urcorner\!\urcorner &= \mathsf{silent}\, \ulcorner P \urcorner \\
\ulcorner\!\ulcorner\!\ulcorner u(v).P \urcorner\!\urcorner\!\urcorner &= \mathsf{in}\ u\ (\lambda v. \ulcorner P \urcorner) \\
\ulcorner\!\ulcorner\!\ulcorner \overline{u}\langle v\rangle.P \urcorner\!\urcorner\!\urcorner &= \mathsf{out}\ u\ v\, \ulcorner P \urcorner
\end{aligned}
$$

In the proof of adequacy of reduction, we will find it useful to refer to the following inverse representation function.

Definition 23. (*Inverse representation function* $\llcorner\_\lrcorner$)

$$
\begin{aligned}
\llcorner \cdot \lrcorner &= 0 \\
\llcorner u\,{:}\,\mathsf{chan}, \Gamma \lrcorner &= \mathsf{new}\, u\,{:}\,\mathsf{chan}\, \llcorner \Gamma \lrcorner \\
\llcorner u\,{:}\,\mathsf{proc}\ P, \Gamma \lrcorner &= !(\llcorner P \lrcorner) \mid \llcorner \Gamma \lrcorner \\
\llcorner x^{\wedge}\mathsf{proc}\ P, \Gamma \lrcorner &= \llcorner P \lrcorner \mid \llcorner \Gamma \lrcorner \\
\llcorner x^{\wedge}\mathsf{choice}\ M, \Gamma \lrcorner &= \llcorner M \lrcorner \mid \llcorner \Gamma \lrcorner
\end{aligned}
$$

$$
\begin{aligned}
\llcorner \mathsf{par}\ P\ Q \lrcorner &= \llcorner P \lrcorner \mid \llcorner Q \lrcorner \\
\llcorner \mathsf{new}\ (\lambda u\,{:}\,\mathsf{chan}.\, P) \lrcorner &= \mathsf{new}\, u\, (\llcorner P \lrcorner) \\
\llcorner \mathsf{rep}\ P \lrcorner &= !(\llcorner P \lrcorner) \\
\llcorner \mathsf{sync}\ M \lrcorner &= \llcorner M \lrcorner
\end{aligned}
$$

$$
\begin{aligned}
\llcorner \mathsf{null} \lrcorner &= 0 \\
\llcorner \mathsf{alt}\ c\ M \lrcorner &= \llcorner c \lrcorner + \llcorner M \lrcorner
\end{aligned}
$$

$$
\begin{aligned}
\llcorner \mathsf{silent}\ P \lrcorner &= \tau.(\llcorner P \lrcorner) \\
\llcorner \mathsf{in}\ u\ (\lambda v\,{:}\,\mathsf{chan}.\, P) \lrcorner &= u(v).(\llcorner P \lrcorner) \\
\llcorner \mathsf{out}\ u\ v\ P \lrcorner &= \overline{u}\langle v \rangle.(\llcorner P \lrcorner)
\end{aligned}
$$

B.2.4 Adequacy of Syntax

Lemma 24. (*Adequacy of the Representation of Syntax*)
Let $\Gamma = u_1\,{:}\,\mathsf{chan}, \ldots, u_n\,{:}\,\mathsf{chan}$.

a. $\Gamma \vdash N \Leftarrow \mathsf{act}$ *iff* $N = \ulcorner c \urcorner$ *where* c *may contain* $u_1 \ldots u_n$.

b. $\Gamma \vdash N \Leftarrow \mathsf{sum}$ *iff* $N = \ulcorner M \urcorner$ *where* M *may contain* $u_1 \ldots u_n$.

c. $\Gamma \vdash N \Leftarrow \mathsf{expr}$ *iff* $N = \ulcorner P \urcorner$ *where* P *may contain* $u_1 \ldots u_n$.

d. $\ulcorner\_\urcorner$ *is a compositional bijection.*

Proof. Standard techniques from LF representation methodology. □

B.2.5 Reduction rules

$$
\begin{array}{lll}
\mathsf{fork} & : & \mathsf{proc\ (par\ P\ Q)} \multimap \{\mathsf{proc\ P} \otimes \mathsf{proc\ Q}\}. \\
\mathsf{name} & : & \mathsf{proc\ (new\ (\lambda u.\, P\ u))} \multimap \{\exists \mathsf{u}\!:\!\mathsf{chan}.\, \mathsf{proc\ (P\ u)}\}. \\
\mathsf{promote} & : & \mathsf{proc\ (rep\ P)} \multimap \{!(\mathsf{proc\ P})\}.
\end{array}
$$

$$\mathsf{choice} \quad : \quad \mathsf{sum} \rightarrow \mathsf{type}.$$

$$\mathsf{suspend} \quad : \quad \mathsf{proc\ (sync\ M)} \multimap \{\mathsf{choice\ M}\}.$$

$$\mathsf{exit} \quad : \quad \mathsf{choice\ null} \multimap \{1\}.$$

$$\mathsf{select} \quad : \quad \mathsf{sum} \rightarrow \mathsf{act} \rightarrow \mathsf{type}.$$

$$
\begin{array}{lll}
\mathsf{this} & : & \mathsf{select\ (alt\ C\ M)\ C}. \\
\mathsf{next} & : & \mathsf{select\ M\ C} \rightarrow \mathsf{select\ (alt\ C'\ M)\ C}.
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{internal} & : & \mathsf{choice\ M} \multimap \mathsf{select\ M\ (silent\ P)} \rightarrow \mathsf{proc\ P}. \\
\mathsf{external} & : & \mathsf{choice\ M_1} \multimap \mathsf{choice\ M_2} \multimap \\
 & & \quad \mathsf{select\ M_1\ (in\ U\ (\lambda w\!:\!chan.\, P\ w))} \rightarrow \mathsf{select\ M_2\ (out\ U\ V\ Q)} \rightarrow \\
 & & \quad \{\mathsf{proc\ (par\ (P\ V)\ Q)}\}.
\end{array}
$$

From this point forward, assume all CLF terms, computations, judgments and derivations are checked with respect to the signature given above.

B.2.6 Representation of Reduction

Definition 25. (*General Contexts* Γ)
The following sorts of context may arise during a computation in CLF.

$$\Gamma ::= . \mid \Gamma, u\!:\!\mathsf{chan} \mid \Gamma, u\!:\!\mathsf{proc}\ P \mid \Gamma, x\,\hat{}\,\mathsf{proc}\ P \mid \Gamma, x\,\hat{}\,\mathsf{choice}\ M$$

Definition 26. (*Representation Relation* $\longleftrightarrow$)
$P \longleftrightarrow \Gamma$ *if and only if* $\llcorner \Gamma \lrcorner = Q$ *and* $Q \equiv P$

Definition 27. (*Context Equivalence* $\Gamma \equiv \Gamma'$)
Let assumptions of the form x?A be either linear or unrestricted assumptions.

$$\frac{}{\Gamma, x?A, y?B, \Gamma' \equiv \Gamma, y?B, x?A, \Gamma'}\ eq_1\ \ (x \notin B, y \notin A)$$

$$\frac{}{\Gamma, \Gamma' \equiv \Gamma, u\!:\!\mathsf{chan}, \Gamma'}\ eq_2\ \ (u \notin \Gamma, \Gamma') \qquad \frac{}{\Gamma, u\!:\!\mathsf{chan}, \Gamma' \equiv \Gamma, \Gamma'}\ eq_3\ \ (u \notin \Gamma, \Gamma')$$

$$\frac{}{\Gamma \equiv \Gamma}\ eq_4 \qquad \frac{\Gamma \equiv \Gamma'' \quad \Gamma'' \equiv \Gamma'}{\Gamma \equiv \Gamma'}\ eq_5$$

Definition 28. (*Composition of Computations* $E\langle E'\rangle$)
We define the composition of two computations E *and* E' *with type* $\top$*, denoted* $E\langle E'\rangle$*, as the computation that results from substituting* E' *for the terminal* $\langle\rangle$ *in* E*.*

Definition 29. (*Representation of Structural Equivalence* $\Longrightarrow_s$)
$\Gamma_1, E \Longrightarrow_s \Gamma_k$ *if*

1. $E = \langle\rangle$ *and* $\Gamma_1 \equiv \Gamma_k$*, or*

2. $E = \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle$ *and there is a normal derivation of the following form:*

$$\dfrac{\cdots \qquad \dfrac{\cdots}{\Gamma_1 \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E' \div \top}}{\Gamma_2 \vdash E' \div \top}$$

and $\Gamma_2, E' \Longrightarrow_s \Gamma_k$ *and* R *is one of the following atomic forms (where we let* x *range over either linear or unrestricted variables):*

$$\mathsf{exit}\hat{\ }x \qquad \mathsf{fork}\hat{\ }x \qquad \mathsf{name}\hat{\ }x \qquad \mathsf{promote}\hat{\ }x \qquad \mathsf{suspend}\hat{\ }x$$

Definition 30. (*Representation of the Single-Step Reduction* $\Rightarrow$)
$\Gamma_0, E\langle\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle\rangle \Rightarrow \Gamma_2'$ *iff* $\Gamma_0, E \Longrightarrow_s \Gamma_1$ *and*

$$\dfrac{\cdots \qquad \overline{\Gamma_1 \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle \div \top}}{\Gamma_2 \vdash \langle\rangle \div \top}$$

and $\Gamma_2 \equiv \Gamma_2'$ *and* R *is one of the following atomic forms (where we let* x*,* x_1*,* x_2 *range over either linear or unrestricted variables):*

$$\mathsf{external}\hat{\ }x\ N \qquad \mathsf{internal}\hat{\ }x_1\hat{\ }x_2\ N_1\ N_2$$

and $N, N_1, N_2 ::= \mathsf{this} \mid \mathsf{next}\ N$

Definition 31. (*Representation of the Multi-Step Reduction* $\Longrightarrow^\star$)
$\Gamma_1, E \Longrightarrow^\star \Gamma_k$ *iff*

1. $\Gamma_1, E \Longrightarrow_s \Gamma_k$

2. $E = E_1\langle E_2\rangle$ *and* $\Gamma_1, E_1 \Rightarrow \Gamma_2$ *and* $\Gamma_2, E_2 \Longrightarrow^\star \Gamma_k$

B.2.7 Properties of Context Equivalence

The following simple properties of contexts and context equivalence will be useful in our proofs of adequacy. In particular, Lemma 35 states that the same structural, single-step and multi-step reductions may occur under any equivalent context.

Lemma 32. (*Context Equivalence Is Symmetric*)
If $\Gamma \equiv \Gamma'$*, then* $\Gamma' \equiv \Gamma$.

Proof. By induction on the derivation $\Gamma \equiv \Gamma'$. □

Lemma 33.
If $\Gamma, u{:}\mathsf{chan}, \Gamma' \vdash E \div \top$ *and* $u \notin \Gamma'$*, then* $u \notin E$.

Proof. By induction on the structure of the normal proofs. □

Lemma 34. (*Frame Lemma, part 1*)
If $\Gamma \equiv \Gamma'$*, then* $\Gamma, \Gamma'' \equiv \Gamma', \Gamma''$.

Proof. By induction on structure the derivation $\Gamma \equiv \Gamma'$. □

Lemma 35.

a. If $\Gamma \vdash E \div \top$ *and* $\Gamma \equiv \Gamma'$*, then* $\Gamma' \vdash E \div \top$.

b. If $\Gamma, E \Longrightarrow_s \Gamma_2$ *and* $\Gamma \equiv \Gamma'$*, then* $\Gamma', E \Longrightarrow_s \Gamma_2$.

c. If $\Gamma, E \Rightarrow \Gamma_2$ *and* $\Gamma \equiv \Gamma'$*, then* $\Gamma', E \Rightarrow \Gamma_2$.

d. If $\Gamma, E \Longrightarrow^\star \Gamma_2$ *and* $\Gamma \equiv \Gamma'$*, then* $\Gamma', E \Longrightarrow^\star \Gamma_2$.

Proof.

a. By induction on the derivation $\Gamma \equiv \Gamma'$.

Case eq_1**:** By exchange property of framework.
Case eq_2**:** By unrestricted weakening property of framework.
Case eq_3**:** By Lemma 33 and strengthening of unused unrestricted assumptions.
Case eq_4**:** Trivial.
Case eq_5**:** : By induction hypothesis.

b,c,d. Corollaries of part (a).

□

B.2.8 Reactive Processes

Rules red_3, red_4 and red_5 allow reduction to occur deep within the structure of a process. The following grammar defines the places where reduction may occur. Lemma 37 formalizes the intuition that reduction may occur in any of these places.

Definition 36. (*Processes with a Hole* H)
$H ::= [] \mid (H \mid P) \mid (P \mid H) \mid \mathsf{new}\, u\!:\!\mathsf{chan}\, H$
We substitute a process Q *for the hole* [] *using the notation* $H[Q]$.

Lemma 37.

a. $\llcorner(\Gamma, \Gamma')\lrcorner = H[\llcorner\Gamma'\lrcorner]$ *for sum processes with a hole* H.

b. If $P \equiv Q$*, then* $H[P] \equiv H[Q]$.

c. If $P \to Q$*, then* $H[P] \to H[Q]$.

d. If $P \longrightarrow^* Q$*, then* $H[P] \longrightarrow^* H[Q]$.

e. If $\llcorner(\Gamma,\Gamma')\lrcorner = H[\llcorner\Gamma'\lrcorner]$, *then* $\llcorner(\Gamma,\Gamma'')\lrcorner = H[\llcorner\Gamma''\lrcorner]$.

Proof.

a. By induction on the structure of Γ.

b. By induction on the structure of H.

c. By induction on the structure of H.

d. By induction on the derivation $P \longrightarrow^* Q$. The base case relies on part *(b)*. The inductive case relies on part *(c)*.

e. By induction on the structure of Γ.

□

A process is *reactive* if it appears in a position allows it to take part in the next step in a computation. In other words, a process is reactive if it appears under "|", "new", or "!", but not if it is prefixed by an input or output action

Definition 38. (*Reactive Processes (1)*)
$Y ::= [] \mid (Y \mid P) \mid (P \mid Y) \mid \mathsf{new}\, u\, Y \mid !Y$

Definition 39. (*Reactive Processes (2)*)
$Z ::= H[Y_1 \mid Y_2]$

We use the notation $Y[M]$ for the process formed by filling the hole in Y with M. We use the notation $Z[M_1, M_2]$ for a process formed by filling the holes in Z with M_1 and M_2 (the hole Y_1 is filled by M_1 and the hole Y_2 is filled by M_2).

Definition 40. (M *in* P)
M *in* P *iff* $P = Y[M']$ *and* $M' \equiv M$

Definition 41. (M_1, M_2 *in* P)
M_1, M_2 *in* P *iff* $M_1 \equiv M_1'$ *and* $M_2 \equiv M_2'$ *and*

1. $P = Z[M_1', M_2']$

2. $P = Z[M_2', M_1']$ *or*

3. $P = H[!Q]$ *and* M_1' *in* Q *and* M_2' *in* Q

Definition 42. (M *in* Γ)
M *in* Γ *iff*

1. $\Gamma = (\Gamma_1, u{:}\mathsf{proc}\ P, \Gamma_2)$ *and* M *in* $\llcorner P\lrcorner$

2. $\Gamma = (\Gamma_1, x^\wedge\mathsf{proc}\ P, \Gamma_2)$ *and* M *in* $\llcorner P\lrcorner$, *or*

3. $\Gamma = (\Gamma_1, x^\wedge\mathsf{choice}\ M', \Gamma_2)$ *and* M *in* $\llcorner M'\lrcorner$

Definition 43. $(M_1, M_2 \; in \; \Gamma)$
$M_1, M_2 \; in \; \Gamma$ *iff*

1. $\Gamma = (\Gamma_1, u{:}\mathsf{proc}\; P, \Gamma_2)$ *and* $M_1 \; in \; \llcorner P \lrcorner$ *and* $M_2 \; in \; \llcorner P \lrcorner$
2. $\Gamma = (\Gamma_1, x^\wedge\mathsf{proc}\; P, \Gamma_2)$ *and* $M_1, M_2 \; in \; \llcorner P \lrcorner$
3. $\Gamma = (\Gamma_1, \Gamma_2)$ *and* $M_1 \; in \; \Gamma_1$ *and* $M_2 \; in \; \Gamma_2$, *or*
4. $\Gamma = (\Gamma_1, \Gamma_2)$ *and* $M_2 \; in \; \Gamma_1$ *and* $M_1 \; in \; \Gamma_2$

Lemma 44.

a. *If* $M \; in \; P$ *and* $P \equiv Q$, *then* $M \; in \; Q$.

b. *If* $M_1, M_2 \; in \; P$ *and* $P \equiv Q$, *then* $M_1, M_2 \; in \; Q$.

Proof. In both parts, by induction on the structure of the derivation of $P \equiv Q$. □

Lemma 45.

a. *If* $M \; in \; \llcorner \Gamma \lrcorner$, *then* $M \; in \; \Gamma$.

b. *If* $M_1, M_2 \; in \; \llcorner \Gamma \lrcorner$, *then* $M_1, M_2 \; in \; \Gamma$.

Proof. By induction on the structure of Γ. □

Lemma 46.

a. *If* $M \; in \; \Gamma$, *then* $\Gamma, E \Longrightarrow_s \Gamma', x^\wedge\mathsf{choice}\; M'$ *and* $\llcorner M' \lrcorner \equiv M$.

b. *If* $M_1, M_2 \; in \; \Gamma$, *then* $\Gamma, E \Longrightarrow_s (\Gamma'', x^\wedge\mathsf{choice}\; M_1', y^\wedge\mathsf{choice}\; M_2')$ *and* $\llcorner M_1' \lrcorner \equiv M_1$ *and* $\llcorner M_2' \lrcorner \equiv M_2$.

Proof.

a. By induction on the nesting depth of $M \; in \; \Gamma$ where the *nesting depth* of an arbitrary M in a context Γ ($depth(M; \Gamma)$) is defined as follows:

$$
\begin{aligned}
depth(M; \Gamma_1, x^\wedge\mathsf{choice}\; M', \Gamma_2) &= 0 \qquad\qquad (if \; \llcorner M' \lrcorner \equiv M) \\
depth(M; \Gamma_1, u{:}\mathsf{proc}\; P, \Gamma_2) &= depth(M; P) + 1 \\
depth(M; \Gamma_1, x^\wedge\mathsf{proc}\; P, \Gamma_2) &= depth(M; P) + 1 \\
depth(M; \mathsf{sync}\; M') &= 1 \qquad\qquad (if \; \llcorner M' \lrcorner \equiv M) \\
depth(M; \mathsf{par}\; P_1\; P_2) &= \min(depth(M; P_1), depth(M; P_2)) + 1 \\
depth(M; \mathsf{new}\,(u{:}\mathsf{chan}\; P)) &= depth(M; P) + 1 \\
depth(M; \mathsf{rep}\; P) &= depth(M; P) + 1
\end{aligned}
$$

b. By induction on $depth(M_1;\Gamma) + depth(M_2;\Gamma)$.

$\square$

Lemma 47.

If $\Gamma \vdash \ulcorner(c_1 + \ldots + c_n + 0)\urcorner \Leftarrow \mathsf{sum}$*, then for all* i*,* $1 \leq i \leq n$*, there exists* N *such that* $\Gamma \vdash N \Leftarrow (\mathsf{select}\ \ulcorner(c_1 + \ldots + c_n + 0)\urcorner\ \ulcorner c_i\urcorner)$.

Proof. By induction on i.

Case $i = 1$**:**

$$\begin{array}{ll} \ulcorner(c_1 + \ldots + c_n + 0)\urcorner = \mathsf{alt}\ \ulcorner c_1\urcorner\ \ulcorner(c_2 + \ldots + c_n + 0)\urcorner & \text{(by definition of } \ulcorner\_\urcorner) \\ \Gamma \vdash \mathsf{this} \Leftarrow \mathsf{select}\ (\mathsf{alt}\ \ulcorner c_1\urcorner\ \ulcorner(c_2 + \ldots + c_n + 0)\urcorner)\ \ulcorner c_1\urcorner & \text{(by type of } \mathsf{this}\text{, well-formedness} \\ & \text{of } \ulcorner(c_1 + \ldots + c_n + 0)\urcorner\ \textit{in}\ \Gamma) \end{array}$$

Hence $N = \mathsf{this}$.

Case $i = k$**:**

$$\begin{array}{ll} \Gamma \vdash N' \Leftarrow \mathsf{select}\ \ulcorner(c_2 + \ldots + c_n + 0)\urcorner\ \ulcorner c_k\urcorner & \text{(by induction)} \\ \Gamma \vdash (\mathsf{next}\ N') \Leftarrow \mathsf{select}\ (\mathsf{alt}\ \ulcorner c_1\urcorner\ \ulcorner(c_2 + \ldots + c_n + 0)\urcorner)\ \ulcorner c_k\urcorner & \text{(by type of next, well-formedness} \\ & \text{of the sequence)} \end{array}$$

Hence $N = \mathsf{next}\ N'$.

$\square$

Lemma 48.

a. *If* $\Gamma \vdash N \Leftarrow (\mathsf{select}\ M\ (\mathsf{silent}\ P))$*, then* $\llcorner M \lrcorner \to \llcorner P \lrcorner$

b. *If* $\Gamma \vdash N_1 \Leftarrow (\mathsf{select}\ M_1\ (\mathsf{in}\ u\ v.P))$ *and if* $\Gamma \vdash N_2 \Leftarrow (\mathsf{select}\ M_2\ (\mathsf{out}\ u\ w.Q))$*, then* $\llcorner(M_1 \mid M_2)\lrcorner \to \llcorner(P[w/v] \mid Q)\lrcorner$.

Proof.

a. By induction on the form of N

$$\begin{array}{rll} \llcorner M \lrcorner & = & c_1 + \ldots + c_n + \tau.\llcorner P \lrcorner + M' \\ & \equiv & \tau.\llcorner P \lrcorner + c_1 + \ldots + c_n + M' \quad \text{(by } str_{14}\text{, } str_{16}) \\ & \longrightarrow & \llcorner P \lrcorner \quad \text{(by } red_1) \end{array}$$

Hence, $\llcorner M \lrcorner \longrightarrow \llcorner P \lrcorner$.

b. By induction on the structure of the normal proof N_1,

$$\begin{array}{rll} \llcorner M_1 \lrcorner & = & c_1 + \ldots + c_n + u(v).\llcorner P \lrcorner + M' \\ & \equiv & u(v).\llcorner P \lrcorner + c_1 + \ldots + c_n + M' \quad \text{(by } str_{14}\text{, } str_{16}). \end{array}$$

By induction on the structure of normal proof N_2,

$$\begin{array}{rll} \llcorner M_2 \lrcorner & = & c'_1 + \ldots + c'_m + \overline{u}\langle w\rangle.\llcorner Q \lrcorner + M'' \\ & \equiv & \overline{u}\langle w\rangle.\llcorner Q \lrcorner + c'_1 + \ldots + c'_m + M'' \quad \text{(by } str_{14}\text{, } str_{16}). \end{array}$$

By red_2, $\llcorner M_1 \lrcorner \mid \llcorner M_2 \lrcorner \to \llcorner(P[w/v] \mid Q)\lrcorner$.

□

Lemma 49.

a. If $M_1 \equiv c + M_2$, then $M_1 = c_1 + \ldots + c_n + 0$ and $c \equiv c_i$ for some i.

b. If $c \equiv \tau.P$, then $c = \tau.P'$ and $P \equiv P'$.

c. If $c \equiv u(v).P$, then $c = u(v).P'$ and $P \equiv P'$.

d. If $c \equiv \overline{u}\langle v\rangle.P$, then $c = \overline{u}\langle v\rangle.P'$ and $P \equiv P'$.

Proof. *(a, b, c, d)* by induction on the structural equivalence relation in the premise. □

Lemma 50.

a. If $\Gamma, E \Longrightarrow_s \Gamma'$ and $\Gamma', E' \Longrightarrow_s \Gamma''$, then $\Gamma, E\langle E'\rangle \Longrightarrow_s \Gamma''$.

b. If $\Gamma, E \Longrightarrow_s \Gamma'$ and $\Gamma', E' \Rightarrow \Gamma''$, then $\Gamma, E\langle E'\rangle \Rightarrow \Gamma''$.

Proof.

a. By induction on the structure of E. Uses Lemma 35*(b)* in the base case.

b. Corollary of part *(a)*.

□

B.2.9 Adequacy of Reductions

In this section, we state and prove our final adequacy results for structural equivalence, single-step and multi-step reductions.

Lemma 51.

a. If $\Gamma \equiv \Gamma'$, then $\llcorner\Gamma\lrcorner \equiv \llcorner\Gamma'\lrcorner$.

b. If $\Gamma, E \Longrightarrow_s \Gamma'$, then $\llcorner\Gamma\lrcorner \equiv \llcorner\Gamma'\lrcorner$.

Proof.

a. By induction on the derivation of context equivalence.

Case eq_1**:** Given: $\Gamma, x?A, y?B, \Gamma' \equiv \Gamma, y?B, x?A, \Gamma'$ $(x \notin B, y \notin A)$
Let X, Y range over assumptions of the following forms:
$u{:}\mathsf{proc}\ P$ $\quad$ $x\hat{}\mathsf{proc}\ P$ $\quad$ $x\hat{}\mathsf{choice}\ M$
Let U, W range over assumptions of the form $u{:}\mathsf{chan}$
There are 4 subcases:

$$\begin{array}{rcll} \llcorner(\Gamma, X, Y, \Gamma')\lrcorner & = & H[Q \mid R \mid \llcorner\Gamma'\lrcorner] & \text{(By definition of } \llcorner\_\lrcorner\text{, Lemma 37}\textit{(a)}\text{)} \\ & = & H[R \mid Q \mid \llcorner\Gamma'\lrcorner] & \text{(By } str_2,\ str_{10}\text{, then Lemma 37}\textit{(b)}\text{)} \\ & = & \llcorner(\Gamma, Y, X, \Gamma')\lrcorner & \text{(By definition of } \llcorner\_\lrcorner\text{, Lemma 37}\textit{(a)}\text{)} \end{array}$$

$\llcorner(\Gamma, U, W, \Gamma')\lrcorner$**:** As above except we use equivalence rule str_6.

$\llcorner(\Gamma, U, X, \Gamma')\lrcorner$**:** As above except we use equivalence rule str_5.

$\llcorner(\Gamma, X, U, \Gamma')\lrcorner$**:** As above except we use equivalence rules str_5 and str_8.

Case eq_2**:** As above except we use equivalence rule str_4.

Case eq_3**:** As above except we use equivalence rules str_4 and str_8.

Case eq_4**:** Trivial.

Case eq_5**:** By induction and then rule str_9.

b. By induction on the structure of the proofs E given via the definition $\Longrightarrow_s$.

□

Corollary 52.
If $\Gamma, E \Longrightarrow_s \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \longleftrightarrow \Gamma'$*.*

Proof. Corollary of Lemma 51*(b)*. □

Theorem 53. (*Adequacy 1*)

a. If $\Gamma, E \Longrightarrow_s \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \longleftrightarrow \Gamma'$*.*

b. If $\Gamma, E \Rightarrow \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \rightarrow Q$ *and* $Q \longleftrightarrow \Gamma'$*.*

c. If $\Gamma, E \Longrightarrow^\star \Gamma'$ *and* $P \longleftrightarrow \Gamma$*, then* $P \longrightarrow^* Q$ *and* $Q \longleftrightarrow \Gamma'$*.*

d. E *is a normal proof iff there exists* Γ *and* Γ' *such that* $\Gamma, E \Longrightarrow^\star \Gamma'$*.*

Proof.

a.
$$\begin{array}{ll} P \equiv \llcorner\Gamma\lrcorner & \text{(by definition of } \longleftrightarrow\text{)} \\ \llcorner\Gamma\lrcorner \equiv \llcorner\Gamma'\lrcorner & \text{(by Lemma 51}\textit{(b)}\text{)} \\ P \longleftrightarrow \llcorner\Gamma'\lrcorner & \text{(by Transitivity of } \equiv \text{ and definition of } \longleftrightarrow\text{)} \end{array}$$

b. By definition, $\Gamma, E\langle \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle\rangle \Rightarrow \Gamma'$ iff

$$\frac{\cdots \qquad \overline{\Gamma_1 \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \langle\rangle \div \top}}{\Gamma_2 \vdash \langle\rangle \div \top}$$

and $\Gamma_2 \equiv \Gamma'$ [6] and either of the following two cases apply

$N = \mathsf{external}^\wedge x\ N'$. Then,

$$
\begin{array}{ll}
P \longleftrightarrow \Gamma & \text{(by assumption)} \\
P \longleftrightarrow \Gamma_1 \qquad [1] & \text{(by adequacy 1a)} \\
\Gamma_1 \equiv \Gamma'', x^\wedge\mathsf{choice}\ M_1 \qquad [2] & \text{(by CLF typing)} \\
\Gamma_2 \equiv \Gamma'', x^\wedge\mathsf{proc}\ P' \qquad [3] & \text{(by CLF typing)} \\
\overline{\Gamma''_1} \vdash N' \Leftarrow (\mathsf{select}\ M_1\ (\mathsf{silent}\ P')) \qquad [4] & \text{(by CLF typing)} \\
\llcorner M_1 \lrcorner \longrightarrow \llcorner P' \lrcorner \qquad [6] & \text{(by [4], L. 13a)}
\end{array}
$$

$$
\begin{array}{rlll}
P & \equiv & \llcorner(\Gamma'', x^\wedge\mathsf{choice}\ M_1)\lrcorner & \text{(by [1,2], def of } \longleftrightarrow) \\
& = & H[\llcorner M_1 \lrcorner] & \text{(by Lemma 37}\mathit{(a)}) \\
& \longrightarrow & H[\llcorner P' \lrcorner] & \text{(by [5], Lemma 37}\mathit{(c)}) \\
& = & \llcorner(\Gamma'', x^\wedge\mathsf{proc}\ P')\lrcorner & \text{(by Lemma 37}\mathit{(e)}) \\
& \equiv & \llcorner \Gamma_2 \lrcorner & \text{(by [3], Lemma 51}\mathit{(a)}) \\
& \equiv & \llcorner \Gamma' \lrcorner & \text{(by [6], Lemma 51}\mathit{(a)})
\end{array}
$$

Hence, $P \rightarrow \llcorner \Gamma' \lrcorner$ and by definition $\llcorner \Gamma' \lrcorner \longleftrightarrow \Gamma'$.

$N = \mathsf{internal}^\wedge x_1{}^\wedge x_2\ N_1\ N_2$. Then,

$$
\begin{array}{ll}
P \longleftrightarrow \Gamma & \text{(by assumption)} \\
P \longleftrightarrow \Gamma_1 & \text{(by Adequacy 1a)} \\
\Gamma_1 \equiv \Gamma'', x^\wedge\mathsf{choice}\ M_1, y^\wedge\mathsf{choice}\ M_2 & \text{(by CLF typing)} \\
\Gamma_2 \equiv \Gamma'', x^\wedge\mathsf{proc}\ (P'[w/v] \mid Q) & \text{(by CLF typing)} \\
\text{and } \overline{\Gamma'_1} \vdash N_1 \Leftarrow \mathsf{select}\ M_1\ (u(v).P') \qquad [1] & \text{(by CLF typing)} \\
\text{and } \overline{\Gamma'_1} \vdash N_2 \Leftarrow \mathsf{select}\ M_2\ (\overline{u}\langle w\rangle.Q) \qquad [2] & \text{(by CLF typing)} \\
\llcorner M_1 \lrcorner \mid \llcorner M_2 \lrcorner \longrightarrow \llcorner(P'[w/v] \mid Q)\lrcorner \qquad [3] & \text{(by [1,2], Lemma 48}\mathit{(b)})
\end{array}
$$

$$
\begin{array}{rlll}
P & \equiv & \llcorner(\Gamma'', x^\wedge\mathsf{choice}\ M_1, y^\wedge\mathsf{choice}\ M_2)\lrcorner & \text{(by definition of } \longleftrightarrow) \\
& = & H[\llcorner M_1 \lrcorner \mid \llcorner M_2 \lrcorner] & \text{(by def of 5a)} \\
& \longrightarrow & H[\llcorner(P'[w/v] \mid Q)\lrcorner] & \text{(by [3], Lemma 5c)} \\
& = & \llcorner(\Gamma'', x^\wedge\mathsf{proc}\ (P'[w/v] \mid Q))\lrcorner & \text{(by Lemma 37}\mathit{(e)}) \\
& \equiv & \llcorner \Gamma_2 \lrcorner & \text{(by Lemma 51}\mathit{(a)}) \\
& \equiv & \llcorner \Gamma' \lrcorner & \text{(by Lemma 51}\mathit{(a)})
\end{array}
$$

Hence, $P \longrightarrow \llcorner \Gamma' \lrcorner$ and $\llcorner \Gamma' \lrcorner \longleftrightarrow \Gamma'$ by definition.

c. By induction on the derivation $\Gamma, E \Longrightarrow^{\star} \Gamma'$.

d. If direction, by induction on the derivation $\Gamma, E \Longrightarrow^{\star} \Gamma'$. Only if direction, by induction on the structure of normal proofs.

$\square$

Theorem 54. (*Adequacy 2*)

a. If $P \equiv Q$ and $P \longleftrightarrow \Gamma$, then $Q \longleftrightarrow \Gamma$.

b. If $P \longleftrightarrow \Gamma$ and $P \longrightarrow Q$, then there exists E and Γ' such that $\Gamma, E \Rightarrow \Gamma'$ and $Q \longleftrightarrow \Gamma'$.

c. If $P \longleftrightarrow \Gamma$ and $P \longrightarrow^{*} Q$, then there exists E and Γ' such that $\Gamma, E \Longrightarrow^{\star} \Gamma'$ and $Q \longleftrightarrow \Gamma'$.

Proof.

a. Trivial by the definition of $\Gamma \longleftrightarrow P$ and transitivity of $\equiv$.

b. By Lemma 22, there exists an F such that $P \equiv F$ and $F \to^- Q'$ and $Q' \equiv Q$. By definition of $\longleftrightarrow$ and transitivity of $\equiv$, our obligation reduces to proving:

   If $F \equiv \llcorner\Gamma\lrcorner$ and $F \to^- Q'$, then there exists E and Γ' such that $\Gamma, E \Rightarrow \Gamma'$ and $Q' \equiv \llcorner\Gamma'\lrcorner$.

   Case 1: $F = \mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid \tau.P + M$

   $Q' = \mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid P$ (By definition of $\to^-$)
   $\tau.P + M\ in\ \llcorner\Gamma\lrcorner$ (By Lemma 44 and $F \equiv \llcorner\Gamma\lrcorner$)
   $\tau.P + M\ in\ \Gamma$ (By Lemma 45)
   $\Gamma, E \Longrightarrow_s \Gamma_1, x^\wedge\mathsf{choice}\ M_1'$ and
   $\tau.P + M \equiv \llcorner M_1'\lrcorner$ (By Lemma 46)
   $\llcorner M_1'\lrcorner = c_1 + \ldots + c_n + \mathsf{0}$ and
   $c_i = \tau.P'$ and
   $P \equiv P'$ (By Lemma 49*(a,b)*)
   $\overline{\Gamma} \vdash N_1 \Leftarrow \mathsf{select}\ M_1'\ \tau.\ulcorner P'\urcorner$ (By Lemma 47)
   $\Gamma_1, x^\wedge\mathsf{choice}\ M_1' \vdash$
   $\mathsf{let}\ \{z^\wedge\mathsf{proc}\ (\ulcorner P'\urcorner)\} = \mathsf{internal}^\wedge x N_1\ \mathsf{in}\ \langle\rangle \Rightarrow \Gamma_1, z^\wedge\mathsf{proc}\ (\ulcorner P'\urcorner)$ (By CLF typing)

   Hence, by definition of $\Rightarrow$, $\Gamma, E\langle\mathsf{let}\ \{z^\wedge\mathsf{proc}\ (\ulcorner P'\urcorner)\} = \mathsf{external}^\wedge x N_1\ \mathsf{in}\ \langle\rangle\rangle \Rightarrow \Gamma_1, z^\wedge\mathsf{proc}\ (\ulcorner P'\urcorner)$.
   $Q \longleftrightarrow \Gamma_1, z^\wedge\mathsf{proc}\ (\ulcorner P'\urcorner)$

   Case 2: $F = \mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid c_1 + M_1' \mid c_2 + M_2'$

   where $c_1 = u(v).P$ and $c_2 = \overline{u}\langle w\rangle.Q''$.
   $Q' = \mathsf{new}\, u_1 \ldots u_n\, !P_1 \mid \ldots \mid !P_m \mid M_1 \mid \ldots \mid M_k \mid P(v) \mid Q''$ (By definition of $\to^-$)
   $c_1 + M_1', c_2 + M_2'\ in\ \llcorner\Gamma\lrcorner$ (By Lemma 44 and $F \equiv \llcorner\Gamma\lrcorner$)
   $c_1 + M_1', c_2 + M_2'\ in\ \Gamma$ (By Lemma 45)
   $\Gamma, E \Longrightarrow_s \Gamma_1, x^\wedge\mathsf{choice}\ M_1'', y^\wedge\mathsf{choice}\ M_2''$ and
   $c_1 + M_1' \equiv \llcorner M_1''\lrcorner$ and
   $c_2 + M_2' \equiv \llcorner M_2''\lrcorner$ (By Lemma 46)
   $\llcorner N_1\lrcorner = d_1 + \ldots + d_n + \mathsf{0}$ and
   $d_i = u(v).P'$ and
   $P \equiv P'$ (By Lemma 49*(a,c)*)
   $\llcorner N_2\lrcorner = d_1 + \ldots + d_m + \mathsf{0}$ and
   $d_j = \overline{u}\langle v\rangle.Q'''$ and
   $Q'' \equiv Q'''$ (By Lemma 49*(a,d)*)
   $\overline{\Gamma} \vdash N_1 \Leftarrow \mathsf{select}\ M_1''\ d_i$ (By Lemma 47)
   $\overline{\Gamma} \vdash N_2 \Leftarrow \mathsf{select}\ M_2''\ d_j$ (By Lemma 47)
   $\Gamma_1, x^\wedge\mathsf{choice}\ M_1'', y^\wedge\mathsf{choice}\ M_2'',$
   $\mathsf{let}\ \{z^\wedge\mathsf{proc}\ (P'(v) \mid Q''')\} = \mathsf{external}^\wedge x^\wedge y N_1 N_2\ \mathsf{in}\ \langle\rangle$
   $\Rightarrow \Gamma_1, z^\wedge\mathsf{proc}\ (P'(v) \mid Q''')$ (By CLF typing)

   Hence, by definition of $\Rightarrow$, $\Gamma, E\langle\mathsf{let}\ \{z^\wedge\mathsf{proc}\ (P(v) \mid Q'')\} = \mathsf{external}^\wedge x^\wedge y N_1 N_2\ \mathsf{in}\ \langle\rangle\rangle \Rightarrow \Gamma_1, z^\wedge\mathsf{proc}\ (P(v)' \mid Q''')$

$Q \longleftrightarrow \Gamma_1, z^\wedge \mathsf{proc}\ (P(v)' \mid Q''')$.

By induction on the structure of the derivation $P \longrightarrow^* Q$ and appeal to part *(b).*

□

References

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[BL93] J. Banatre and D. LeMetayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[BM00] Roberto Bruni and Ugo Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1–2):46–89, 2000.

[BM01] David Basin and Seán Matthews. Logical frameworks. In Dov Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition, 2001. In preparation.

[BMMS98] R. Bruni, J. Meseguer, U. Montanari, and V. Sassone. A comparison of Petri net semantics under the collective token philosophy. *Springer-Verlag LNCS 1538*, pages 225–244, 1998.

[CDL+99] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In P. Syverson, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, June 1999. IEEE Computer Society Press.

[CDL+00] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 35–51, Cambrige, UK, July 2000. IEEE Computer Society Press.

[Cer95] Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M. I. Sessa, editors, *Proceedings of the 1995 Joint Conference on Declarative Programming — GULP-PRODE'95*, pages 313–318, Marina di Vietri, Italy, 1995.

[Cer01a] Iliano Cervesato. A specification language for crypto-protocol based on multiset rewriting, dependent types and subsorting. In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Proceedings of the Workshop on Specification, Analysis and Validation for Emerging Technologies — SAVE'01*, pages 1–22, Paphos, Cyprus, 1 December 2001.

[Cer01b] Iliano Cervesato. Typed MSR: Syntax and examples. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Proceedings of the First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security — MMM'01*, pages 159–177, St. Petersburg, Russia, 21–23 May 2001. Springer-Verlag LNCS 2052.

[CP98] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 1998. To appear in a special issue with invited papers from LICS'96, E. Clarke, editor.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[EW90] U. Engberg and G. Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *Proceedings of the 15th Annual Colloquium on Trees in Algebra and Programming*, pages 147–161, Copenhagen, Denmark, 1990. Springer-Verlag LNCS 431.

[GG90] V. Gehlot and C. Gunter. Normal process representatives. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 200–207, Philadelphia, PA, 1990. IEEE Computer Society Press.

[Hal85] Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[IP98] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[Jen97] Kurt Jensen. A brief introduction to coloured Petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems, proceeding of the TACAS'97 Workshop*, pages 203–208, Enschede, The Netherlands, 1997. Springer-Verlag LNCS 1217.

[Maz95] Antoni W. Mazurkiewicz. True versus artificial concurrency. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, pages 53–68, Warsaw, Poland, 1995. Chapman & Hall.

[Mil92] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic. *Mathematical Structures in Computer Science*, 1:66–101, 1991. Revised version of paper in LNCS 389.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe01a] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, MA, June 2001. IEEE Computer Society Press.

[Pfe01b] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001. In press.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Rei85] W. Reisig. *Petri Nets, an Introduction.* ETACS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[Rep99] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[VC00] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear LF. Draft paper, September 2000.

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