A Concurrent Logical Framework (Extended Abstract)

Iliano Cervesato
iliano@itd.nrl.navy.mil

AES Division
ITT Industries, Inc.
Alexandria, VA 22303-1410

Frank Pfenning*
fp@cs.cmu.edu

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

David Walker
dpw@cs.princeton.edu

Computer Science Department
Princeton University
Princeton, NJ, 08544

Kevin Watkins
kw@cs.cmu.edu

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

January 15, 2002

Synopsis. We introduce CLF, a 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 $\top$. Because CLF is intuitionistic and dependent, it has internal objects representing concurrent computations.

The monadic encapsulation and a novel, algorithmic formulation of the underlying type theory lead to a tractable notion of definitional equality that captures true concurrency: computations that differ only in the order of independent steps are indistinguishable.

We illustrate the expressive power of the framework through encodings of Petri nets and Milner's synchronous π-calculus. We also have similarly elegant encodings of ML with futures, lazy evaluation, and concurrency primitives in the style of CML, and of the security protocol specification language MSR, but they have been omitted in this extended abstract.

*Corresponding author. Phone: +1 412 268-6343, Fax: +1 412 268-3608.

1 Introduction

A logical framework [?, ?] is a meta-language for the specification and implementation of deductive systems, which are used pervasively in logic and the theory of programming languages. A logical framework should be as simple 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 [?] which originated the use of dependent types. It was followed by LF [?], 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 [?].

Representations of systems involving state remained cumbersome until the design of the linear logical framework LLF [?] and its close relative RLF [?]. LLF is a conservative extension of LF with the linear function type $A \multimap B$, the additive product type $A \,\&\, B$, and the additive unit type $\top$. For example, LLF allows an elegant representation of Mini-ML with mutable references that reifies imperative computations as objects. Properties of computations then become properties of objects in the framework. The type theory underlying LLF corresponds to the largest freely generated fragment of intuitionistic linear logic [?, ?] 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 [?, ?]. We also refer to LLF as the *asynchronous fragment* of intuitionistic linear logic, since it contains precisely those connectives that are asynchronous in the terminology of Andreoli [?] and Howe [?].

While LLF solves many problems associated with stateful computation, the encoding of *concurrent computations* remained unsatisfactory for several reasons. One of the problems is that LLF formulations of concurrent systems inherently sequentialize the computation steps. A related issue is that the framework cannot distinguish between the don't-care non-determinism of concurrent execution and the don't-know non-determinism of logic programming.

In this paper we develop CLF, a conservative extension of LLF with intrinsic support for concurrency. Concurrent, don't-care non-deterministic computations are encapsulated in a monad [?], which provides a clear separation from the don't-know non-determinism already present in LLF. Furthermore, the definitional equality on computations inside the monad identifies different interleavings of independent steps, thereby expressing *true concurrency* [?].

We illustrate the framework's expressive power through several sample encodings. The representation of the synchronous π-calculus reprises some of the basic techniques from LF for variable binding and scope and shows different interacting forms of non-determinism. The encoding of Petri nets demonstrates true concurrency in computation traces.

Besides the design of a concurrent logical framework and the presentation of some sample applications, our paper makes several other contributions. It captures, in a monad, the differences between Andreoli's synchronous and asynchronous connectives [?] in the intuitionistic setting [?, ?] and thereby arrives at a tractable equational theory between proofs. Unlike Moggi's original formulation of the monadic meta-language [?] no commuting conversions arise at all, unless one chooses to include the equations that internalize true concurrency.

Finally, we present a formulation of a logical framework with dependencies originating entirely from the notion of canonical form. Type checking and conversion to canonical form are described via terminating algorithms as advocated by de Bruijn [?], which leaves definitional equality as a derived notion. This elegant style of presentation of a type theory is particularly appropriate for a logical framework, where the notion of canonical form is paramount.

The remainder of the paper is organized as follows. In Section 2 we survey some of the related work. We then present the concurrent logical framework in several layers in Section 3, followed

by the encoding of Petri nets in Section 4 and the π-calculus in Section 5. We sketch the basic properties of CLF in Section 6 and conclude in Section 7 with some remarks on future work.

The accompanying technical reports [?, ?] amplify the discussion here with a full description of the meta-theoretic properties of CLF, the proofs of adequacy for the representations discussed here, and several additional fully-developed representations of Concurrent ML and the security protocol specification framework MSR [?]. The representation of Concurrent ML illustrates the use of dependent types and a combination of sequential and concurrent computation. It also introduces a presentation of programming language semantics using destination-passing style which, in concert with the available framework features, allows highly modular and concise presentation of many constructs, including eager evaluation, lazy evaluation, futures, and synchronous communication in the style of CML.

2 Related Work

Right from its inception, linear logic [?] has been advocated as a logic with an intrinsic notion of state and concurrency (see [?, ?, ?, ?, ?], among others). In the literature, many connections between concurrent calculi and linear logic have been observed. To mention just a few most closely related to this paper, we have representations of Petri nets in linear logic [?, ?], translations between the π-calculus and linear logic [?, ?] and formulations of action calculi in intuitionistic linear logic [?, ?].

In a logical framework, we remove ourselves by one degree from the actual semantics; we represent rather than embed calculi. Thereby, CLF provides another point of view on many of the investigations listed above. For example, it is possible to give a representation of classical linear logic and the π-calculus in CLF. We conjecture that it is also possible to formalize translations between them as proposed, for example, by Miller [?] and Bellin and Scott [?], although this is left to future work.

Most closely related to our work is Miller's logical framework Forum [?] which is based on a sequent calculus for classical linear logic and focusing proofs [?]. As shown by Miller and elaborated by Chirimar [?], Forum can also represent concurrency. Our work extends Forum in several important directions. First, it is a type theory based on natural deduction and therefore offers an internal notion of proof object which is not available in Forum. Among other things, this means we can represent relations on deductions and therefore on concurrent computations. Second, implementations of Forum as a logic programming language as originally envisioned have proved to be very difficult because the don't-know non-determinism and backtracking of logic programming stand in conflict with the don't-care non-determinism and scheduling of concurrent programming. In CLF, concurrency (and therefore don't-care non-determinism) is encapsulated in the monad, which provides a clear boundary between the two operational principles. While design and implementation of an operational semantics for CLF is left to future work, we claim it provides a more promising basis.

As already mentioned above, CLF is a conservative extension of LLF with the asynchronous connectives $\otimes$, $\mathbf{1}$, $!$, and $\exists$, encapsulated in a monad. The idea of monadic encapsulation goes back to Moggi's monadic meta-language [?, ?] and is used heavily in functional programming. Our formulation follows the judgmental presentation of Pfenning and Davies [?] that completely avoids the need for commuting conversions, but treats neither linearity nor the existence of normal forms. This permits us to reintroduce some equations to model true concurrency in a completely orthogonal fashion. The exploration of monads in logic programming by Bekkers and Tarau [?] concentrates on the use of monads for data structures and all-solution predicates which is quite different from our application and concerned neither with additional logical connectives nor a true extension of the operational semantics. Benton and Wadler [?] explore the relationship of Moggi's monadic meta-language and term calculi for linear logic with Benton's adjoint calculus, which bears some intriguing similarities with CLF, but is not a type theory and does not identify the logical connectives inherited

from lax logic and linear logic as we do here.

The method of defining a type theory by a typed operational semantics goes back to the Automath languages [?] and has been applied to LF by Felty [?]. We significantly extend and streamline the ideas behind Felty's *canonical* LF and its extension to LLF [?] via a simple inductive definition of *canonical substitutions*, which avoid explicit reference to β-normalization. This idea seems quite robust; in particular it works directly for the monad and the encapsulated operators.

3 The Concurrent Logical Framework CLF

In this section we present a brief overview of the concurrent logical framework (CLF). The design of CLF builds on relatively few, but deep ideas, which means we have to rely rather extensively on previous work. We hope the introduction here will be sufficient to appreciate the examples. The interested reader is referred to a forthcoming technical report [?] for further details.

We present the system in several layers, each containing some of the ideas necessary to appreciate the complete system. A summary of the rules and connectives can be found in Appendix A.

3.1 The Logical Framework LF

The type theory underlying the logical framework LF [?] is based on the dependent type constructor Π. We give here a new formulation, similar to Felty's canonical LF [?], in which only canonical forms are well-typed. The relevant notion of *canonical form* is a β-normal, η-long form, because in the methodology of LF it is always the canonical forms that are in bijective correspondence with the expressions and deductions of the encoded object language. Moreover, we erase type labels on λ-abstractions, since such type labels are redundant for the checking of canonical forms [?].

We mark those syntactic categories that will later be extended with $\mid \ldots$; the others change only insofar as the types, kinds, or objects contained in them are extended.

$$
\begin{array}{rcl}
\text{Normal Kinds} & K & ::= \mathsf{type} \mid \Pi u{:}A.\, K \\
\text{Atomic Types} & P & ::= a \mid P\ N \\
\text{Normal Types} & A & ::= \Pi u{:}A_1.\, A_2 \mid P \mid \ldots \\
\text{Atomic Objects} & R & ::= c \mid u \mid R\ N \mid \ldots \\
\text{Normal Objects} & N & ::= \lambda u.\, N \mid R \mid \ldots \\
\text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, u{:}A \\
\text{Signatures} & \Sigma & ::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A
\end{array}
$$

We write a for type family constants and c for object constants, both declared in signatures Σ with their kind and type, respectively. Variables u are declared in contexts with their type. We make the uniform assumption that no constant or variable may be declared more than once in a signature or context, respectively. We also allow tacit renaming of variables bound by $(\Pi u{:}A.)$ and $(\lambda u.)$. As usual, we avoid an explicit non-dependent function type by thinking of $A \to B$ as an abbreviation for $\Pi u{:}A.\, B$ where u does not occur in B.

From the point of view of natural deduction, atomic objects are composed of destructors corresponding to elimination rules, while normal objects are built from constructors corresponding to introduction rules. The typing rules are now *bi-directional* which mirrors the syntactic structure of normal forms: we check a normal object against a type, and we synthesize a type for an atomic object. We write $U \Leftarrow V$ to indicate that U is checked against a given V (which we assume is valid), and $U \Rightarrow V$ to indicate that U synthesizes a V (which we prove is valid).

$\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind}$ $\quad K$ is a valid kind

$\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}$ $\quad A$ is a valid type

$\Gamma \vdash_\Sigma P \Rightarrow K$ $\quad P$ is atomic of kind K

$\Gamma \vdash_\Sigma N \Leftarrow A$ $\quad N$ is normal of type A

$\Gamma \vdash_\Sigma R \Rightarrow A$ $\quad R$ is atomic of type A

Since the signature Σ does not change in any of the rules for types and objects, we henceforth omit it from the inference rules for the sake of brevity. The main difficulty in these judgments arises in the rule for application, since a substitution has to be performed due to the nature of dependent function types. We write $[N/u{:}A^-]^{\mathrm{a}}B$ for the normal type that corresponds, informally, to the canonical form of $[N/u]B$, and similarly for $[N/u{:}A^-]^{\mathrm{k}}K$. This form of *canonical substitution* is a technical innovation in this paper; we postpone its explanation to Section 6. It is presented as an inductively defined partial function, guaranteeing termination and hence the decidability of all our typing judgments. We write $P' \equiv P$ for the checking of definitional equality, which at this stage is syntactic equality modulo α-conversion.

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

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

$$\frac{}{\Gamma \vdash a \Rightarrow \Sigma(a)}\ a \qquad \frac{\Gamma \vdash P \Rightarrow \Pi u{:}A.\, K \quad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash P\ N \Rightarrow [N/u{:}A^-]^{\mathrm{k}}K}\ \Pi\mathbf{KE}$$

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

$$\frac{}{\Gamma \vdash c \Rightarrow \Sigma(c)}\ c \qquad \frac{}{\Gamma \vdash u \Rightarrow \Gamma(u)}\ u \qquad \frac{\Gamma \vdash R \Rightarrow \Pi u{:}A.\, B \quad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash R\ N \Rightarrow [N/u{:}A^-]^{\mathrm{a}}B}\ \Pi\mathbf{E}$$

We omit the straightforward rules for checking the validity of signatures or contexts (see Appendix A).

3.2 The Linear Logical Framework LLF

The linear logical framework [**?**] is a conservative extension of LF with a linear function type $A \multimap B$, additive pairs $A \,\&\, B$ and additive unit $\top$. Note that unlike RLF [**?**], linear functions cannot be dependent. The new formulation of LF purely in terms of canonical forms, bi-directional checking, and canonical substitution extends to a new formulation of LLF. In order to extend our judgments to these new types, we add linear variables x and linear hypotheses of the form $x \hat{:} A$. Because linear functions are never dependently typed, we can segregate all the assumptions into two zones, written $\Gamma; \Delta$, where the assumptions $u{:}A$ in Γ are unrestricted as in LF, and the assumptions $x \hat{:} A$ in Δ are linear. Linear assumptions can never depend on each other and can be freely permuted; taking advantage of this, we write Δ_1, Δ_2 for any decomposition of a linear context into two parts without duplication. Unrestricted assumptions also can never depend on linear assumptions, because types cannot contain free linear variables. The formation judgments for types thus also depend only on the unrestricted assumptions.

$$
\begin{array}{rrl}
\text{Normal Types} & A & ::= \ldots \mid A_1 \multimap A_2 \mid A_1 \,\&\, A_2 \mid \top \mid \ldots \\
\text{Atomic Objects} & R & ::= \ldots \mid x \mid R \hat{} N \mid \pi_1 R \mid \pi_2 R \mid \ldots \\
\text{Normal Objects} & N & ::= \ldots \mid \hat{\lambda} x.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \ldots \\
\text{Linear Contexts} & \Delta & ::= \cdot \mid \Delta, x \hat{:} A
\end{array}
$$

The inference rules introduce no particular difficulty and maintain their bi-directional nature. We only show the modified judgments:

$$
\begin{array}{ll}
\Gamma; \Delta \vdash_\Sigma N \Leftarrow A & N \text{ is normal of type } A \\
\Gamma; \Delta \vdash_\Sigma R \Rightarrow A & R \text{ is atomic of type } A
\end{array}
$$

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

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

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

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

$$
\frac{}{\Gamma; x \hat{:} A \vdash x \Rightarrow A}\, x
$$

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

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

We do not show here the straightforward modification of the rules from LF to accommodate linear assumptions (see Appendix A).

LLF can be seen as the type theory corresponding to the largest freely generated fragment of intuitionistic linear logic in which all right rules are invertible (also called *asynchronous* [**?**]). For previous formulations of LLF this means that the notion of canonical form remains well-defined and unique canonical forms exist [**?**, **?**]; here, it is the reason the typing rules from previous formulations of LLF can be directly adopted into our bi-directional system.

3.3 The Monadic Framework

The next step will be to add a monadic type constructor so we can distinguish computations from objects, isolating effects. Just as in Moggi's original proposal [**?**, **?**] we remain uncommitted, for the moment, as to which particular form of effect is encapsulated by the monad. Since we would like to preserve the uniqueness of canonical forms, we introduce a new syntactic class of *expressions* E that correspond to computations. Thus the monadic framework is the extension to the linear, dependent case of a formulation proposed by Davies and the second author [**?**].

$$\begin{array}{rrl} \text{Normal Types} & A & ::= \ldots \mid \{A\} \\ \text{Normal Objects} & N & ::= \ldots \mid \{E\} \\ \text{Expressions} & E & ::= \mathsf{let}\ \{x\} = R\ \mathsf{in}\ E \mid N \end{array}$$

From the logical point of view, the type $\{A\}$ corresponds to the proposition $\bigcirc A$ from a linear variant of lax logic [**?**], with $\{E\}$ as the proof object for $\bigcirc$-introduction and $\mathsf{let}\ \{x\} = R\ \mathsf{in}\ E$ as the proof object for $\bigcirc$-elimination. The restriction to atomic objects R means that no redices of the form $\mathsf{let}\ \{x\} = \{E'\}\ \mathsf{in}\ E$ are allowed. The introduction of a separate category of expressions and its associated new typing judgment $\Gamma; \Delta \vdash_\Sigma E \leftarrow A$ avoid the need for commuting conversions. Their function is taken over by a new form of substitution [**?**] based on an analysis of the object being substituted rather than the object being substituted into. Further details are presented in Section 6.

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

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

$$\frac{\Gamma; \Delta_1 \vdash R \Rightarrow \{A_0\} \quad \Gamma; \Delta_2, x \hat{:} A_0 \vdash E \leftarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash (\mathsf{let}\ \{x\} = R\ \mathsf{in}\ E) \leftarrow A}\ \{\}\mathbf{E} \qquad \frac{\Gamma; \Delta \vdash N \Leftarrow A}{\Gamma; \Delta \vdash N \leftarrow A}\ \Leftarrow\leftarrow$$

At present we have no examples that employ the monadic operator in isolation, but judging from its important role in functional programming, such applications are plausible.

3.4 The Concurrency Monad

Next, we introduce a new syntactic category of *synchronous* types S, but allow them only inside the monad, that is, we replace $\{A\}$ by $\{S\}$. With the exception of 0 and $S_1 \oplus S_2$ (which are well-behaved, but the presentation of which is left to future work), the synchronous types correspond to the synchronous connectives of intuitionistic linear logic [**?**]. In addition, we have to introduce two new syntactic classes to model the introduction and elimination rules for the synchronous connectives. These are the *monadic objects* M, which correspond to the introductions, and *monadic patterns* p, which are needed to express the eliminations.

$$\begin{array}{rrl} \text{Normal Types} & A & ::= \ldots \mid \{S\} \\ \text{Synchronous Types} & S & ::= S_1 \otimes S_2 \mid 1 \mid \exists u{:}A.\, S \mid\ !A \mid A \\ \text{Expressions} & E & ::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\ \text{Monadic Objects} & M & ::= M_1 \otimes M_2 \mid 1 \mid [N, M] \mid\ !N \mid N \\ \text{Monadic Patterns} & p & ::= p_1 \otimes p_2 \mid 1 \mid [u, p] \mid\ !u \mid x \end{array}$$

The synchronous connectives have invertible left rules in the sequent calculus. This means we can always apply the left rules for these connectives in any order without losing completeness. Since such proofs should be indistinguishable, Andreoli introduces an ordered context L of assumptions that is used like a stack, fixing the order of eliminations, an idea adapted to the intuitionistic case by Howe [**?**]. We introduce a corresponding ordered context Ψ and a judgment that breaks down a pattern into the assumptions on the individual variables contained in them, $\Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S$.

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

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

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

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

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

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

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

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

$$\frac{\Gamma; \cdot \vdash N \Leftarrow A \quad \Gamma; \Delta \vdash M \Leftarrow [N/u{:}A^-]^{\mathrm{s}} S}{\Gamma; \Delta \vdash [N, M] \Leftarrow \exists u{:}A.\, S}\ \exists\mathbf{I} \qquad \frac{\Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \cdot \vdash\, !N \Leftarrow\, !A}\ !\mathbf{I}$$

Note that these rules require a further substitution, $[N/u{:}A^-]^{\mathrm{s}}S$ which can be thought of intuitively as returning the canonical form of $[N/u]S$. Despite the rather complete set of connectives for linear logic, which would usually entail a large number of commuting conversions and a complex equational theory, the language above requires only α-conversion as its definitional equality. This is because the bi-directional typing rules themselves enforce normal forms, and the monadic encapsulation prevents the usual commuting conversions from being applicable. The decidability of this type theory and the "admissibility of cut" are claimed in Section 6.

3.5 The Concurrent Logical Framework CLF

The language from the previous section is quite expressive and has many desirable meta-theoretic properties. For many kinds of applications, some of which are discussed in the remainder of this paper, it is fully adequate. As we will see, computations in concurrent calculi such as Petri nets and the π-calculus are naturally interpreted with an *interleaving semantics*. For example, in a concrete Petri net where two independent transitions can fire, there will be two *different* computations, one for each possible first step.

However, ideally, our language should be *truly concurrent* so that we cannot observe the order of independent events. Perhaps surprisingly, this can be achieved by a very simple device. We extend our notion of definitional equality from α-conversion to encompass the reflexive, transitive, and congruent closure generated by all equations of the form

$$(\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ (\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E)) \equiv_{\mathrm{c}} (\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ (\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E))$$

where R_1 and R_2 are *independent*: no variable bound in p_1 is free in R_2 and no variable bound in p_2 is free in R_1. (Technically, it is more convenient to define $\equiv_c$ as a simultaneous congruence with bi-simulation on expressions—we omit the formal definition here since it can be seen to coincide with the one above.)

In the only rule requiring definitional equality, we now replace α-conversion by α-conversion modulo the concurrency equations.

$$\frac{\Gamma; \Delta \vdash R \Rightarrow P' \quad P' \equiv_c P}{\Gamma; \Delta \vdash R \Leftarrow P} \Rightarrow\Leftarrow$$

We now have re-introduced a commuting conversion into the calculus, but in a very controlled manner. For example, there is no case corresponding to the conversion

$$(\mathsf{let}\ x \otimes y = M_1\ \mathsf{in}\ M_2)\ M_3 = (\mathsf{let}\ x \otimes y = M_1\ \mathsf{in}\ M_2\ M_3)$$

in ordinary linear λ-calculus, because the monadic encapsulation prevents

$$\{\mathsf{let}\ \{x \otimes y\} = R\ \mathsf{in}\ E\}\ N = \{\mathsf{let}\ \{x \otimes y\} = R\ \mathsf{in}\ E\ N\}$$

from being well-typed.

The CLF type theory remains decidable with this truly concurrent notion of definitional equality because checking of the new definitional equality continues to be decidable. The admissibility of cut also continues to hold—see Section 6 for details.

This concludes the presentation of the rules of CLF. Its syntax and typing rules are summarized in Appendix A. We now illustrate the use of this calculus through several examples.

4 Petri Nets and True Concurrency

Encodings of Petri nets in linear logic have been well-known for some time [**?**, **?**, **?**]; here we show how such representations can be expressed in CLF. We furthermore examine the structure of the computations as objects in CLF. We concentrate on examples, since space does not permit a formal treatment of Petri nets and their encoding in the framework, which can be found in [**?**].

A *Petri net* is defined by a collection of *places*, *transitions*, *arcs*, and *tokens*. Every transition has input arcs and output arcs that connect it to places. The system evolves by changing the tokens in various places according to the following rules.

1. A transition is enabled if every place connected to it by an input arc contains at least one token.
2. We non-deterministically select one of the enabled transitions in a net to fire.
3. A transition fires by removing one token from each input place and adding one token to each output place of the transition.

Slightly more generally, there may be multiple arcs between a place and a transition, which either generates or consumes multiple tokens. Multiple arcs are represented by an arc labeled by its multiplicity n. As an example, consider the producer/consumer net in Figure 1.

We represent every place by a CLF constant of type place. A token on place p is represented by a linear assumption of the form $x{:}\mathsf{tok}\ p$. If there are several tokens, there are several such assumptions. Every transition is represented by a corresponding linear implication into the monad, with the tokens at the origins of the input arcs as (curried) antecedents and the tokens at the ends of the output arcs as the succedents.

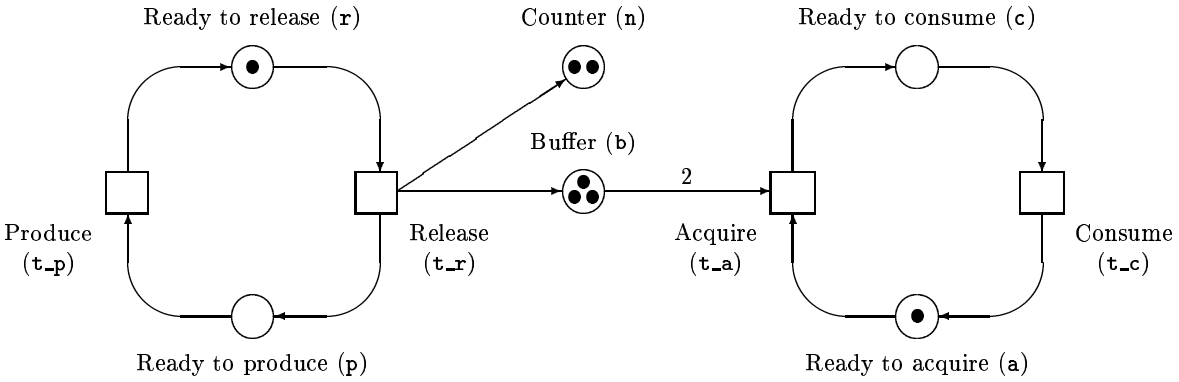

Figure 1: A producer/consumer net

place	:	type.	tok	:	place → type.
r, p, n, b, c, a	:	place.	t_p	:	tok p ⊸ {tok r}.
			t_r	:	tok r ⊸ {tok p ⊗ tok b ⊗ tok n}.
			t_a	:	tok b ⊸ tok b ⊸ tok a ⊸ {tok c}.
			t_c	:	tok c ⊸ {tok a}.

The reading of the transitions is quite intuitive. For example, t_r says: "*if there is a state with a token on place* r, *we can transition to a state with an additional token on* p, b, *and* n, *thereby consuming the token on* r".

The shown initial state is represented by

$$x_1 \hat{:} \text{tok r},\ x_2 \hat{:} \text{tok n},\ x_3 \hat{:} \text{tok n},\ x_4 \hat{:} \text{tok b},\ x_5 \hat{:} \text{tok b},\ x_6 \hat{:} \text{tok b},\ x_7 \hat{:} \text{tok a}\ .$$

A computation starting with the release transition t_r is represented by the expression

$$\text{let } \{x_8 \otimes x_9 \otimes x_{10}\} = \text{t\_r} \hat{} x_1 \text{ in E}\ ,$$

where E represents the rest of the computation starting in state

$$x_2 \hat{:} \text{tok n},\ x_3 \hat{:} \text{tok n},\ x_4 \hat{:} \text{tok b},\ x_5 \hat{:} \text{tok b},\ x_6 \hat{:} \text{tok b},\ x_7 \hat{:} \text{tok a},\ x_8 \hat{:} \text{tok p},\ x_9 \hat{:} \text{tok b},\ x_{10} \hat{:} \text{tok n}\ .$$

A computation starting with the acquire transition t_a is represented by the expression

$$\text{let } \{x_8\} = \text{t\_a} \hat{} x_4 \hat{} x_5 \hat{} x_7 \text{ in E}'\ ,$$

where E′ represents the rest of the computation starting in state

$$x_1 \hat{:} \text{tok r},\ x_2 \hat{:} \text{tok n},\ x_3 \hat{:} \text{tok n},\ x_6 \hat{:} \text{tok b},\ x_8 \hat{:} \text{tok c}\ .$$

Note there are five other isomorphic computations representing a firing of t_a, consuming x_4 and x_6 or x_6 and x_5, etc., instead of x_4 and x_5. The two initial steps above are independent. Therefore, their composition can be represented as two nested let-expressions in either order.

$$\text{let } \{x_8 \otimes x_9 \otimes x_{10}\} = \text{t\_r} \hat{} x_1 \text{ in let } \{x_{11}\} = \text{t\_a} \hat{} x_4 \hat{} x_5 \hat{} x_7 \text{ in E}''$$
$$\text{let } \{x_{11}\} = \text{t\_a} \hat{} x_4 \hat{} x_5 \hat{} x_7 \text{ in let } \{x_8 \otimes x_9 \otimes x_{10}\} = \text{t\_r} \hat{} x_1 \text{ in E}''$$

These two expressions are definitionally equal in the framework. In other words, they are identified by a concurrency equation. This means that in CLF we have no means to observe which step was taken first and therefore that we have true concurrency.

The reader is referred to [**?**] for a formal treatment including detailed proofs of adequacy. This encoding is easily extended to more expressive languages such as colored Petri nets [**?**] and formalisms based on multiset rewriting such as, e.g., the security protocol specification framework MSR [**?**].

5 The π-Calculus

The π-calculus [**?**] was designed 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 the synchronous π-calculus can be represented in CLF. Due to space constraints we must assume the reader is already familiar with the π-calculus.

Syntax. The syntax of the π-calculus consists of channels (u, v, w), process expressions (P, Q, R) and sums (M), the latter modeling synchronization and communication.

$$\begin{array}{rrl} \text{Process Expressions} & P & ::= 0 \mid (P \mid Q) \mid \mathsf{new}\, u\, P \mid\, !P \mid M \\ \text{Sums} & M & ::= \tau.P \mid u(v).P \mid \overline{u}\langle v\rangle.P \mid M_1 + M_2 \end{array}$$

We represent the three syntactic classes of the π-calculus with three CLF types. The process expressions themselves are represented as objects of type expr, sums by objects of type sum. We represent every channel u by a corresponding unrestricted variable u:chan of the same name.

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

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

The representation function $\ulcorner\cdot\urcorner$ maps process expressions into CLF objects with type expr and $\urcorner\cdot\urcorner$ maps sums into CLF objects with type sum. It uses higher-order abstract syntax [**?**] to represent the bound variable v in the restriction $\mathsf{new}\, v\, P$ and in the input process $u(v).P$.

$$\begin{array}{rcl} \ulcorner 0 \urcorner &=& \mathsf{null} \\ \ulcorner P \mid Q \urcorner &=& \mathsf{par}\ \ulcorner P \urcorner\ \ulcorner Q \urcorner \\ \ulcorner \mathsf{new}\, u\, P \urcorner &=& \mathsf{new}\ (\lambda u.\ulcorner P \urcorner) \\ \ulcorner !P \urcorner &=& \mathsf{rep}\ \ulcorner P \urcorner \\ \ulcorner M \urcorner &=& \mathsf{sync}\ \urcorner M \urcorner \end{array} \qquad \begin{array}{rcl} \urcorner \tau.P \urcorner &=& \mathsf{silent}\ \ulcorner P \urcorner \\ \urcorner u(v).P \urcorner &=& \mathsf{in}\ u\ (\lambda v.\ulcorner P \urcorner) \\ \urcorner \overline{u}\langle v\rangle.P \urcorner &=& \mathsf{out}\ u\ v\ \ulcorner P \urcorner \\ \urcorner M_1 + M_2 \urcorner &=& \mathsf{alt}\ \urcorner M_1 \urcorner\ \urcorner M_2 \urcorner \end{array}$$

Operational Semantics. The operational semantics of the π-calculus consists of two parts. First, a *structural congruence relation* $P \equiv Q$ divides the set of processes into congruence classes. A second *reaction relation* $P \longrightarrow Q$ describes how actual computation occurs. Two central rules describe the silent transition and how input and output processes interact:

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

Three further rules makes reaction insensitive to the choice of representative within a structural congruence class, and allow communication to occur under a name binder and in parallel with other processes.

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

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:chan and replicable processes r:proc P, while Δ contains $x\hat{?}$proc Q for all other processes active in the initial state. The goal type $\top$ allows the computation to stop at any point, modeling a 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 0 will be interpreted as the CLF connective 1 and the parallel composition | 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 enough of the structural congruences associated with parallel composition in order to obtain an adequate encoding (modulo structural congruences) of embedded process expressions. Eventually, the process expression will be decomposed into a sum, at which point we have to synchronize. Synchronization non-deterministically selects either a silent action, or a pair of matching input and output actions from two different sums. We summarize the resulting signature.

```
proc    : expr → type.

exit    : proc null ⊸ {1}.
fork    : ΠP:expr. ΠQ:expr. proc (par P Q) ⊸ {proc P ⊗ proc Q}.
name    : ΠP:chan → expr. proc (new (λu. P u)) ⊸ {∃u:chan. proc (P u)}.
promote : ΠP:expr. proc (rep P) ⊸ {!(proc P)}.
```

The elimination of the exponential ! puts an unrestricted assumption r:proc P into the context, allowing arbitrarily many subsequent uses of r. 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.

```
choice  : sum → type.
suspend : ΠM:sum. proc (sync M) ⊸ {choice M}.
```

Finally, we come to the reaction rules, which 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. This is in contrast to Miller's encoding [**?**], which does not make this important distinction, interpreting synchronous choice directly as additive conjunction &.

```
select  : sum → sum → type.

choose1 : ΠM1:sum. ΠM2:sum. ΠN:sum. select M1 N → select (alt M1 M2) N.
choose2 : ΠM1:sum. ΠM2:sum. ΠN:sum. select M2 N → select (alt M1 M2) N.
quiet   : ΠP:expr. select (silent P) (silent P).
input   : ΠU:chan. ΠP:chan → expr. select (in U (λv. P v)) (in U (λv. P v)).
output  : ΠU:chan. ΠV:chan. ΠP:expr. select (out U V P) (out U V P).
```

The second family selects the guarded processes to react and operates on them to perform the actual reaction step. For an internal 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.

$$
\begin{array}{lcl}
\mathsf{internal} & : & \Pi \mathsf{M}{:}\mathsf{sum}.\,\Pi \mathsf{P}{:}\mathsf{expr}.\,\mathsf{choice\ M} \multimap \mathsf{select\ M\ (silent\ P)} \to \{\mathsf{proc\ P}\} \\
\mathsf{external} & : & \Pi \mathsf{M}{:}\mathsf{sum}.\,\Pi \mathsf{N}{:}\mathsf{sum}.\,\Pi \mathsf{U}{:}\mathsf{chan}.\,\Pi \mathsf{W}{:}\mathsf{chan}.\,\Pi \mathsf{P}{:}\mathsf{chan} \to \mathsf{expr}.\,\Pi \mathsf{Q}{:}\mathsf{expr}. \\
 & & \quad \mathsf{choice\ M} \multimap \mathsf{choice\ N} \multimap \\
 & & \quad \mathsf{select\ M\ (in\ U\ (}\lambda \mathsf{v}.\,\mathsf{P\ v))} \to \mathsf{select\ N\ (out\ U\ W\ Q)} \to \\
 & & \quad \{\mathsf{proc\ (P\ W)} \otimes \mathsf{proc\ Q}\}.
\end{array}
$$

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 for $P{:}\mathsf{chan} \to \mathsf{expr}$ is carried out. This is a minor variant of a standard technique in logical frameworks.

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 π-calculus and linear logic. Other encodings are possible, and some are discussed in [**?**], to which the reader is also referred for proofs of adequacy.

6 Properties of CLF

Our new presentation of the frameworks LF and LLF and our new monadic framework, the new concurrency monad, and CLF all depend on a simple inductively defined notion of *canonical substitutions*. Canonical substitution is used in the typing judgments for the framework when the argument of a dependent function needs to be substituted into the function's result type. Since this framework is redex-free by its very presentation, a theorem analogous to the *admissibility of cut* familiar from the sequent calculus, which is also redex-free, assumes a much greater importance. Since the admissibility of cut is witnessed by the canonical substitutions, the admissibility theorem amounts to the statement that canonical substitution is defined on well-typed terms, and that the result of such a substitution is well typed.

Intuitively, the canonical substitution operator $[N_0/x{:}\tau]^{\mathrm{n}}N$ substitutes the normal object N_0 into the normal object N, replacing occurrences of the variable x in N. However, since variable occurrences are atomic objects, and there is no coercion in the system from normal objects to atomic objects (which would create a redex), our substitution must go beyond the ordinary syntactic substitution $[N_0/x]N$ to perform the computations ordinarily relegated to a notion of β-reduction on redices. The termination of this computation phase is witnessed by the *simple type* τ which appears as an additional argument to the substitution. Simple types are generated from dependent types by a straightforward erasure function A^- presented in Appendix A. The syntax of the dependent type constructors changes to reflect that dependencies have been eliminated:

$$(\Pi u{:}A.\,B)^- = A^- \to B^- \qquad (\exists u{:}A.\,S)^- = A^- \mathbin{!\otimes} S^- \qquad (P\ N)^- = P^-$$

Concretely, substitution is a partial function $[N_0/x{:}\tau]^{\mathrm{n}}N$. It is inductively defined by a nested induction, where the outer induction is over the simple type τ and the inner induction is over the pair (N_0, N). The induction order for τ is the syntactic containment order $\tau < \tau'$, while for the pair we say $(N_0, N) < (N_0', N')$ if either $N_0 < N_0'$ and $N = N'$ or else $N_0 = N_0'$ and $N < N'$. For substitution, no distinction is made between the two classes of variables u and x; we use x for either. We also need the concepts of atomic and principal substitutions, discussed below.

$$
\begin{array}{rl}
\text{Normal Substitution} & [N_0/x{:}\tau]^{\mathrm{n}}N = N' \\
\text{Atomic Substitution} & [N_0/x{:}\tau]^{\mathrm{r}}R = R' \\
\text{Principal Substitution} & [N_0/x{:}\tau]^{\beta}R = (N' : \tau')
\end{array}
$$

The substitution of a normal object into an atomic object falls into two cases depending on whether the variable at the head of the atomic object is the substitution variable, or a different variable. If the latter, the structure of the atomic object is left in place and the substitution only applies congruences. Otherwise, the atomic object will decompose the normal object by a process similar to β-reduction—we refer to this as principal substitution. The result of the decomposition is a normal object together with its simple type (used to enforce termination). We present here only the definition for the LF fragment and object level. The remainder may be found in Appendix A.

$$[N_0/x{:}\tau]^{\mathrm{n}}(\lambda u.\, N) = \lambda u.\, [N_0/x{:}\tau]^{\mathrm{n}} N$$
$$[N_0/x{:}\tau]^{\mathrm{n}}(R) = [N_0/x{:}\tau]^{\beta} R \text{ or } [N_0/x{:}\tau]^{\mathrm{r}} R$$

$$[N_0/x{:}\tau]^{\mathrm{r}}(c) = c$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(y) = y \text{ provided } x \neq y$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(R\ N) = ([N_0/x{:}\tau]^{\mathrm{r}} R)\ ([N_0/x{:}\tau]^{\mathrm{n}} N)$$

$$[N_0/x{:}\tau]^{\beta}(x) = (N_0 : \tau)$$
$$[N_0/x{:}\tau]^{\beta}(R\ N) = ([[N_0/x{:}\tau]^{\mathrm{n}} N/u{:}\tau_1]^{\mathrm{n}} N' : \tau_2)$$
$$\text{if } [N_0/x{:}\tau]^{\beta} R = (\lambda u.\, N' : \tau_1 \to \tau_2)$$
$$\text{and } \tau_1 < \tau\ .$$

We can see that this final case of substitution terminates: the first substitution $[N_0/x{:}\tau]^{\beta} R$ operates on N_0 and the subterm $R < R\ N$ at the same simple type τ; the inner substitution operates on N_0 and $N < R\ N$ at the same simple type τ; and finally, the outer substitution operates on the essentially arbitrarily large objects $[N_0/x{:}\tau]^{\mathrm{n}} N$ and N', but only at the strictly smaller type $\tau_1 < \tau$. Since the outer induction is on the simple type while the inner induction is on the pair of objects, all these recursions are allowable.

A few items are perhaps worth noting. First, a principal substitution $[N_0/x{:}\tau]^{\beta} R$ is only defined if the head variable of R is x, while the atomic substitution $[N_0/x{:}\tau]^{\mathrm{r}} R$ is only defined if the head is not x. Thus, there is no ambiguity in the definition of normal substitution when it comes to a coerced atomic object. Second, the condition $\tau_1 < \tau$ expresses that τ_1 is a subterm of τ. It is an easy theorem that the condition will always hold, since each case of principal substitution decomposes a type further into one of its subformulas. The condition (and others like it) is nevertheless incorporated into the definition of substitution to make it transparently inductive, hence terminating. The fact that substitution terminates and definitional equality is decidable leads directly to the decidability of typing in the framework. Decidability is clear for the definitional equality based on α-conversion, and it holds for the truly concurrent equality as well.

Theorem 1 (Decidability of truly concurrent equality) *It is decidable whether any instance of the equation $N_1 \equiv_{\mathrm{c}} N_2$ holds.*

This result is most easily seen by a syntax-directed variant presentation of the truly concurrent equality using simultaneous congruence and bi-simulation on expressions [**?**].

Theorem 2 (Decidability of CLF) *It is decidable whether any instance of the judgment $\Gamma; \Delta \vdash N \Leftarrow A$ holds.*

Proof: Once the induction hypothesis is strengthened to include the decidability of all the other typing judgments on which normal object typing depends, the result follows by a simple induction over the structure of the derivation of the judgment concerned, making use of the termination of substitution and the decidability of definitional equality. ■

Furthermore, the presence of the simple type itself as an argument to substitution is redundant if the objects involved are known to be well typed, since the simple types generated as principal substitution decomposes a well-typed object will always be what they are required to be. An

optimized type checker can take advantage of this by staging the process of type checking so that it is an invariant that whenever a substitution is applied the objects involved are known to be well typed. The fact that this is possible is evident upon examination of the flow of information through the typing rules.

Finally, although the cases of substitution presented above are all "rightist" in that they are syntax-directed based on the object to the right of the substitution, there are a few dissident "leftist" substitutions, written $\langle E_0/p{:}\varsigma\rangle^{\mathrm{e}}E$ and $\langle M_0/p{:}\varsigma\rangle^{\mathrm{m}}E$, which are syntax-directed based on the object at the left of the substitution. This idea already appears in previous work on the monadic connective [**?**]. These "leftist" substitutions motivate the component of the induction order mentioning the term being substituted.

With canonical substitution in hand, we can go on to prove the admissibility of cut for CLF. This theorem remains agnostic as to the framework's notion of definitional equality, so it applies equally to CLF both with and without the concurrency equations of Section 3.5. The only assumptions made about definitional equality are transitivity and stability under substitution, which are clear for α-conversion, and claimed for the concurrent equality [**?**].

Theorem 3 (Admissibility of cut)
If $\Gamma_{\mathrm{L}};\cdot \vdash N_0 \Leftarrow A$ *is derivable,* $\Gamma_{\mathrm{L}}, u{:}A, \Gamma_{\mathrm{R}}; \Delta \vdash N \Leftarrow C$ *is derivable, and the substitutions* $[N_0/u{:}A^-]^{\mathrm{a}}\Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$, $[N_0/u{:}A^-]^{\mathrm{a}}\Delta = \Delta'$ *and* $[N_0/u{:}A^-]^{\mathrm{a}}C = C'$ *are defined, the following hold:*

1. ***Progress.*** *The substitution* $[N_0/u{:}A^-]^{\mathrm{n}}N$ *is defined.*
2. ***Preservation.*** *The judgment* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}; \Delta' \vdash [N_0/u{:}A^-]^{\mathrm{n}}N \Leftarrow C'$ *is derivable.*

If $\Gamma; \Delta_1 \vdash N_0 \Leftarrow A$ *and* $\Gamma; \Delta_2, x{:}^{\wedge}A \vdash N \Leftarrow C$ *are derivable, the following hold:*

1. ***Progress.*** *The substitution* $[N_0/x{:}A^-]^{\mathrm{n}}N$ *is defined.*
2. ***Preservation.*** *The judgment* $\Gamma; \Delta_1, \Delta_2 \vdash [N_0/x{:}A^-]^{\mathrm{n}}N \Leftarrow C$ *is derivable.*

This result is mutually dependent on many other admissibility theorems for the other syntactic categories. Space considerations preclude the incorporation of the proof here. Many other important properties of the framework appear as corollaries of the admissibility theorem—the details may be found in the accompanying technical report [**?**].

7 Conclusion

In this paper, we have presented the basic design of a logical framework that internalizes parametric and hypothetical judgments, linear hypothetical judgments, and true concurrency. This supports representation of a wide variety of concepts related to logic and computation in a natural and concise manner. It also poses a host of new questions.

Operational Semantics of CLF. One of the practically important features of the linear logical framework is its operational interpretation as a logic programming language using goal-directed proof search [**?**, **?**]. We conjecture that CLF supports a conservative extension of this operational semantics, where proof search inside the monad proceeds in a concurrent, don't-care non-deterministic manner, while proof search outside the monad proceeds in a don't-know non-deterministic manner using backtracking as in LLF. We have already constructed a representation of Mini-ML with concurrency and parallelism anticipating such an interpretation [**?**]. We plan to investigate such a semantics and relate it to other proposals for combining forward and backward search [**?**] or concurrent computation based on focusing in classical linear logic such as Forum [**?**], LinLog [**?**], and LO [**?**]. One of the novel aspects of such an operational semantics will be unification modulo concurrency equations, extending the prior work on unification in the linear logical framework [**?**].

Properties of Computations. Concurrent computations in an object language are internalized as monadic expressions in CLF. The framework allows type families indexed by objects containing such expressions, which means it is possible to formulate properties of concurrent computations and relations between them. Examples are safety and possibly liveness properties, bi-simulations, and other translations between models of computations. We plan to investigate such representations in the framework and their properties. One critical item here will be to consider a co-inductive interpretation of computations.

Additive Disjunction and Falsehood. At present, the concurrency monad encapsulates all synchronous connectives except for the additive disjunction and falsehood. This would introduce case- and abort-expressions into CLF. Based on the completeness of focusing for first-order intuitionistic logic, we conjecture that the system retains its uniformity and elegance under the addition of these operators. Besides some questions of concrete syntax, the principal complication appears to be a slightly less obvious notion of definitional equality with respect to the concurrency equations. Additive disjunction and falsehood appear to allow a new form of concurrency where processes operate in separate branches of a proof tree and communicate via logic variables.

Case Studies and Applications. Besides the examples presented in this paper, we have also concluded an encoding of a version of ML that simultaneously supports functions, recursion, definitions, pairs, unit type, sum types, void type, recursive types, parametric polymorphic types, intersection types, suspensions with memoization, mutable references, futures in the style of Multilisp [**?**], and concurrency in the style of CML [**?**]. We further have a representation of the first author's security protocol specification framework MSR [**?**]. Other targets for case studies in the realm of concurrent and imperative languages abound and are left to the reader's imagination.

Acknowledgments

We would like to thank Mike Mislove for exchanges regarding the nature of true concurrency.

A Syntax and Judgments of CLF

Syntax.

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

Typing.

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash a \Rightarrow \Sigma(a)}\ a \qquad \frac{\Gamma \vdash P \Rightarrow \Pi u{:}A.K \quad \Gamma;\cdot \vdash N \Leftarrow A}{\Gamma \vdash P\ N \Rightarrow [N/u{:}A^-]^{\mathrm{k}}K}\ \Pi\mathbf{KE}$$

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

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

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

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

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

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

$$\frac{\Gamma;\Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma;\Delta_2 \vdash N \Leftarrow A}{\Gamma;\Delta_1,\Delta_2 \vdash R\hat{}N \Rightarrow B}\ \multimap\mathbf{E} \qquad \frac{\Gamma;\Delta \vdash R \Rightarrow \Pi u{:}A.\,B \quad \Gamma;\cdot \vdash N \Leftarrow A}{\Gamma;\Delta \vdash R\ N \Rightarrow [N/u{:}A^-]^{\mathrm{a}}B}\ \Pi\mathbf{E}$$

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

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

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

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

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

$$\frac{\Gamma;\cdot \vdash N \Leftarrow A \quad \Gamma;\Delta \vdash M \Leftarrow [N/u{:}A^-]^{\mathrm{s}}S}{\Gamma;\Delta \vdash [N,M] \Leftarrow \exists u{:}A.\,S}\ \exists\mathbf{I} \qquad \frac{\Gamma;\cdot \vdash N \Leftarrow A}{\Gamma;\cdot \vdash !N \Leftarrow !A}\ !\mathbf{I}$$

Simple Types.

$$\begin{aligned}
\kappa &::= \mathsf{type}^- \mid \tau \to \kappa \\
\tau &::= a^- \mid \tau_1 \multimap \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \top^- \mid \{\varsigma\} \\
\varsigma &::= \varsigma_1 \otimes \varsigma_2 \mid 1^- \mid \tau\ !\!\otimes \varsigma \mid\ !\tau \mid \tau
\end{aligned}$$

The erasure operators $(K)^- = \kappa$, $(A)^- = \tau$, $(P)^- = a^-$ and $(S)^- = \varsigma$ are defined as follows.

$$\begin{aligned}
(\mathsf{kind})^- &= \mathsf{kind}^- \\
(\mathsf{type})^- &= \mathsf{type}^- \\
(\Pi u{:}A.\,K)^- &= A^- \to K^- \\
(A \multimap B)^- &= A^- \multimap B^- \\
(\Pi u{:}A.\,B)^- &= A^- \to B^- \\
(A \,\&\, B)^- &= A^- \,\&\, B^- \\
(\top)^- &= \top^- \\
(\{S\})^- &= \{S^-\}
\end{aligned}
\qquad
\begin{aligned}
(a)^- &= a^- \\
(P\ N)^- &= P^- \\
(S_1 \otimes S_2)^- &= S_1^- \otimes S_2^- \\
(1)^- &= 1^- \\
(\exists u{:}A.\,S)^- &= A^- \,!\!\otimes S^- \\
(!A)^- &= !(A^-)
\end{aligned}$$

Canonical Substitution. All the substitution operators are presented as a set of mutual partial inductive definitions. The definitions are based on an outer induction over the simple type τ or ς, as the case may be, and an inner induction over the pair $(N_0, \mathcal{C})$ for the structural substitutions $[N_0/x{:}\tau]^{\mathrm{c}}\mathcal{C}$, or (N_0, R) for the principal substitution $[N_0/x{:}\tau]^{\beta}R$, or (E_0, E) for the expression substitution $\langle E_0/p{:}\varsigma\rangle^{\mathrm{e}}E$, or (M_0, E) for the pattern substitution $\langle M_0/p{:}\varsigma\rangle^{\mathrm{m}}E$. The order on simple types is by syntactic containment, while for the pairs it is a simultaneous order derived from the syntactic containment order on the elements of the pair, as explained in Section 6. The mutual

references always respect this order, though in some cases termination is forced by an explicit condition $\tau < \tau'$ which must hold for the substitution to be defined.

$$[N_0/x{:}\tau]^{\mathrm{k}}(\mathsf{type}) = \mathsf{type}$$
$$[N_0/x{:}\tau]^{\mathrm{k}}(\Pi u{:}A.\,K) = \Pi u{:}([N_0/x{:}\tau]^{\mathrm{a}}A).\,([N_0/x{:}\tau]^{\mathrm{k}}K)$$

$$[N_0/x{:}\tau]^{\mathrm{a}}(A \multimap B) = ([N_0/x{:}\tau]^{\mathrm{a}}A) \multimap ([N_0/x{:}\tau]^{\mathrm{a}}B)$$
$$[N_0/x{:}\tau]^{\mathrm{a}}(\Pi u{:}A.\,B) = \Pi u{:}([N_0/x{:}\tau]^{\mathrm{a}}A).\,([N_0/x{:}\tau]^{\mathrm{a}}B)$$
$$[N_0/x{:}\tau]^{\mathrm{a}}(A \,\&\, B) = ([N_0/x{:}\tau]^{\mathrm{a}}A) \,\&\, ([N_0/x{:}\tau]^{\mathrm{a}}B)$$
$$[N_0/x{:}\tau]^{\mathrm{a}}(\top) = \top$$
$$[N_0/x{:}\tau]^{\mathrm{a}}(\{S\}) = \{[N_0/x{:}\tau]^{\mathrm{s}}S\}$$
$$[N_0/x{:}\tau]^{\mathrm{a}}(P) = [N_0/x{:}\tau]^{\mathrm{p}}P$$

$$[N_0/x{:}\tau]^{\mathrm{p}}(a) = a$$
$$[N_0/x{:}\tau]^{\mathrm{p}}(P\ N) = ([N_0/x{:}\tau]^{\mathrm{p}}P)\ ([N_0/x{:}\tau]^{\mathrm{n}}N)$$

$$[N_0/x{:}\tau]^{\mathrm{s}}(S_1 \otimes S_2) = ([N_0/x{:}\tau]^{\mathrm{s}}S_1) \otimes ([N_0/x{:}\tau]^{\mathrm{s}}S_2)$$
$$[N_0/x{:}\tau]^{\mathrm{s}}(1) = 1$$
$$[N_0/x{:}\tau]^{\mathrm{s}}(\exists u{:}A.\,S) = \exists u{:}([N_0/x{:}\tau]^{\mathrm{a}}A).\,([N_0/x{:}\tau]^{\mathrm{s}}S)$$
$$[N_0/x{:}\tau]^{\mathrm{s}}(!A) = !([N_0/x{:}\tau]^{\mathrm{a}}A)$$
$$[N_0/x{:}\tau]^{\mathrm{s}}(A) = [N_0/x{:}\tau]^{\mathrm{a}}A$$

$$[N_0/x{:}\tau]^{\mathrm{n}}(\hat{\lambda}y.\,N) = \hat{\lambda}y.\,([N_0/x{:}\tau]^{\mathrm{n}}N)$$
$$[N_0/x{:}\tau]^{\mathrm{n}}(\lambda u.\,N) = \lambda u.\,([N_0/x{:}\tau]^{\mathrm{n}}N)$$
$$[N_0/x{:}\tau]^{\mathrm{n}}(\langle N_1, N_2\rangle) = \langle [N_0/x{:}\tau]^{\mathrm{n}}N_1, [N_0/x{:}\tau]^{\mathrm{n}}N_2\rangle$$
$$[N_0/x{:}\tau]^{\mathrm{n}}(\langle\rangle) = \langle\rangle$$
$$[N_0/x{:}\tau]^{\mathrm{n}}(\{E\}) = \{[N_0/x{:}\tau]^{\mathrm{e}}E\}$$
$$[N_0/x{:}\tau]^{\mathrm{n}}(R) = [N_0/x{:}\tau]^{\beta}R \text{ or } [N_0/x{:}\tau]^{\mathrm{r}}R$$

$$[N_0/x{:}\tau]^{\mathrm{r}}(c) = c$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(y) = y \text{ provided } x \neq y$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(R\hat{}N) = ([N_0/x{:}\tau]^{\mathrm{r}}R)\hat{}([N_0/x{:}\tau]^{\mathrm{n}}N)$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(R\ N) = ([N_0/x{:}\tau]^{\mathrm{r}}R)\ ([N_0/x{:}\tau]^{\mathrm{n}}N)$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(\pi_1 R) = \pi_1([N_0/x{:}\tau]^{\mathrm{r}}R)$$
$$[N_0/x{:}\tau]^{\mathrm{r}}(\pi_2 R) = \pi_2([N_0/x{:}\tau]^{\mathrm{r}}R)$$

if $[N_0/x{:}\tau]^{\beta}R = \{E'\} : \{\varsigma'\}$ and $\varsigma' < \tau$ then
$[N_0/x{:}\tau]^{\mathrm{e}}(\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) = \langle E'/p{:}\varsigma'\rangle^{\mathrm{e}}\,[N_0/x{:}\tau]^{\mathrm{e}}E$; otherwise,
$[N_0/x{:}\tau]^{\mathrm{e}}(\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) = (\mathsf{let}\ \{p\} = [N_0/x{:}\tau]^{\mathrm{r}}R\ \mathsf{in}\ [N_0/x{:}\tau]^{\mathrm{e}}E)$.
$$[N_0/x{:}\tau]^{\mathrm{e}}(M) = [N_0/x{:}\tau]^{\mathrm{m}}M$$

$$[N_0/x{:}\tau]^{\mathrm{m}}(M_1 \otimes M_2) = ([N_0/x{:}\tau]^{\mathrm{m}}M_1) \otimes ([N_0/x{:}\tau]^{\mathrm{m}}M_2)$$
$$[N_0/x{:}\tau]^{\mathrm{m}}(1) = 1$$
$$[N_0/x{:}\tau]^{\mathrm{m}}([N, M]) = [[N_0/x{:}\tau]^{\mathrm{n}}N, [N_0/x{:}\tau]^{\mathrm{m}}M]$$
$$[N_0/x{:}\tau]^{\mathrm{m}}(!N) = !([N_0/x{:}\tau]^{\mathrm{n}}N)$$
$$[N_0/x{:}\tau]^{\mathrm{m}}(N) = [N_0/x{:}\tau]^{\mathrm{n}}N$$

$[N_0/x{:}\tau]^{\beta}(x) = (N_0 : \tau)$
$[N_0/x{:}\tau]^{\beta}(R \hat{} N) = ([[N_0/x{:}\tau]^{\mathrm{n}} N/y{:}\tau_1]^{\mathrm{n}} N' : \tau_2)$
  if $[N_0/x{:}\tau]^{\beta} R = (\hat{\lambda} y.\, N' : \tau_1 \multimap \tau_2)$ and $\tau_1 < \tau$.
$[N_0/x{:}\tau]^{\beta}(R\ N) = ([[N_0/x{:}\tau]^{\mathrm{n}} N/u{:}\tau_1]^{\mathrm{n}} N' : \tau_2)$
  if $[N_0/x{:}\tau]^{\beta} R = (\lambda u.\, N' : \tau_1 \to \tau_2)$ and $\tau_1 < \tau$.
$[N_0/x{:}\tau]^{\beta}(\pi_1 R) = (N_1 : \tau_1)$
  if $[N_0/x{:}\tau]^{\beta} R = (\langle N_1, N_2 \rangle : \tau_1 \,\&\, \tau_2)$.
$[N_0/x{:}\tau]^{\beta}(\pi_2 R) = (N_2 : \tau_2)$
  if $[N_0/x{:}\tau]^{\beta} R = (\langle N_1, N_2 \rangle : \tau_1 \,\&\, \tau_2)$.

$\langle(\mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ E_0)/p{:}\varsigma\rangle^{\mathrm{e}} E = (\mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ \langle E_0/p{:}\varsigma\rangle^{\mathrm{e}} E)$
$\langle M_0/p{:}\varsigma\rangle^{\mathrm{e}} E = \langle M_0/p{:}\varsigma\rangle^{\mathrm{m}} E$

$\langle M_1 \otimes M_2/p_1 \otimes p_2{:}\varsigma_1 \otimes \varsigma_2\rangle^{\mathrm{m}} E = \langle M_2/p_2{:}\varsigma_2\rangle^{\mathrm{m}}\, \langle M_1/p_1{:}\varsigma_1\rangle^{\mathrm{m}} E$
$\langle 1/1{:}1^{-}\rangle^{\mathrm{m}} E = E$
$\langle [N, M]/[u, p]{:}\tau\ !\otimes\ \varsigma\rangle^{\mathrm{m}} E = \langle M/p{:}\varsigma\rangle^{\mathrm{m}}\, [N/u{:}\tau]^{\mathrm{e}} E$
$\langle !N/!u{:}!\tau\rangle^{\mathrm{m}} E = [N/u{:}\tau]^{\mathrm{e}} E$
$\langle N/x{:}\tau\rangle^{\mathrm{m}} E = [N/x{:}\tau]^{\mathrm{e}} E$