A Concurrent Logical Framework: the Propositional Fragment

Kevin Watkins[1], Iliano Cervesato[2], Frank Pfenning[1], and David Walker[3]

[1] kw,fp@cs.cmu.edu, Department of Computer Science, Carnegie Mellon University
[2] iliano@itd.nrl.navy.mil, Naval Research Laboratory, Washington, D.C.
[3] dpw@cs.princeton.edu, Department of Computer Science, Princeton University

Abstract. We present the propositional fragment CLF_0 of the Concurrent Logical Framework (CLF). CLF extends the Linear Logical Framework to allow the natural representation of concurrent computations in an object language. The underlying type theory uses monadic types to segregate values from computations. This separation leads to a tractable notion of definitional equality that identifies computations differing only in the order of execution of independent steps. From a logical point of view our type theory can be seen as a novel combination of lax logic and dual intuitionistic linear logic. An encoding of a small Petri net exemplifies the representation methodology, which can be summarized as "*concurrent computations as monadic expressions*".

1 Introduction

A logical framework [Pfe01] 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 is determined by an underlying meta-logic or type theory and a methodology for representing the judgments and derivations of interest.

The particular lineage of logical frameworks to which the present paper belongs started with the Automath languages [dB80] followed by LF [HHP93], crystallizing the representation principles of "*judgments as types*" and "*deductions as objects*". LF is based on a minimal type theory with only the dependent function type constructor Π. The representation methodology expresses hypothetical judgments and hypothetical deductions that formalize reasoning from assumptions can be expressed concisely and elegantly using the notion of function space provided by the underlying type theory. In particular, we can achieve a bijection between hypothetical deductions of a judgment and canonical objects of the corresponding type.

Representations in this style of systems involving state remained cumbersome until the design of the linear logical framework LLF [CP98] and its close relative RLF [IP98]. LLF is a conservative extension of LF with selected constructs from linear logic. The representation principles behind LLF are "*state as linear hypotheses*" and "*imperative computations as linear functions*". Again, we can achieve a bijection between imperative computations of a program and canonical objects of the appropriate type.

While LLF solves many problems associated with the representation of imperative computations, in LLF the encoding of *concurrent computations* remains unsatisfactory. In particular, there is no bijection between concurrent computations and canonical objects of appropriate type because the representation inherently sequentializes the computation steps. In other words, we can only represent all interleavings of potentially concurrent steps, but not true concurrency [Maz95].

The present paper presents the propositional fragment of CLF, a conservative extension of LLF with intrinsic support for true concurrency. Expressions representing concurrent computations are encapsulated in a monad [Mog89], thereby preserving the desirable properties of LF and LLF. The definitional equality on expressions inside the monad makes representations of concurrency adequate by ensuring different interleavings of concurrent steps are indistinguishable. Although monads have been used to separate pure and effectful computations in functional programming languages, to the authors' knowledge this is their first use in a logical framework or theorem proving environment to separate one logic from another.

The present paper also presents a new methodology for developing the meta-theory of LF-style logical frameworks. The definition of *canonical forms* for LF objects is of paramount importance because encodings are normally proved adequate by establishing a bijection between computations and canonical objects. The new methodology emphasizes the central role of canonical forms by restricting the framework's syntax so that only canonical objects are well-formed. Type-directed substitution and η-expansion algorithms preserving the canonical forms property are defined.

Though the dependently typed variant of CLF has already been developed [WCPW02], the present discussion is restricted to the propositional fragment CLF_0, which already exhibits the principal phenomena concerning concurrency. The use of the framework is illustrated by an encoding of Petri-net computations because of their simplicity, but related representations for the π-calculus, Concurrent ML, and other languages with concurrency have also been devised [CPWW02]. The representation technique for all of these examples can be summarized as "*concurrent computations as monadic expressions*".

The remainder is organized as follows. Section 2 defines CLF_0, including its syntax and typing rules. It also introduces the Petri-net example and contrasts encodings in LLF_0 and CLF_0 to highlight their strengths and weaknesses. Section 3 outlines the most important meta-theoretic properties of CLF_0, including the definitions of type-directed canonizing substitution and η-expansion. Finally, Section 4 presents a more detailed picture of related work in the area.

2 Propositional CLF

We begin by introducing the propositional fragment of the concurrent logical framework in stages. In the first stage, we briefly review the linear logical framework, its properties, and its shortcomings with respect to concurrency.

2.1 The Linear Fragment

The propositional fragment LLF_0 of the linear logical framework is based on unrestricted and linear hypothetical judgments $\Gamma; \Delta \vdash_\Sigma M : A$ where Γ is a context of unrestricted hypotheses $u{:}A$ (subject to exchange, weakening, and contraction), Δ is a context of linear hypotheses $x\hat{:}A$ (subject only to exchange), M is an object and A is a type. The signature Σ declares the base types and constants from which objects are constructed. Under the Curry-Howard isomorphism, M can also be read as a proof term, and A as a proposition of intuitionistic linear logic in its formulation as DILL [Bar96].

Since the signature is fixed for a given typing derivation, we henceforth suppress it for the sake of brevity. In addition, syntactic objects are considered only up to α-equivalence of their bound variables. Exchange is not noted explicitly in the typing rules, and only instances of the typing rules for which all variables in the contexts have unique names are allowed.

The LF methodology establishes a bijection between *canonical objects* of appropriate type and the terms and deductions of an object language to be represented. The appropriate notion of "canonical" turns out to be long $\beta\eta$-normal forms. In order to define these inductively, the single typing judgment $\Gamma; \Delta \vdash M : A$ is split into two:

$\Gamma; \Delta \vdash N \Leftarrow A$	N is canonical of type A
$\Gamma; \Delta \vdash R \Rightarrow A$	R is atomic of type A

A canonical object N is an introduction form or of base type, in which case it must be atomic. An atomic object R is a sequence of elimination forms applied to a variable or constant. The basic principles associated with these judgments are identity and substitution. From the logical point of view, they show that entailment is reflexive and transitive. From the type-theoretic point of view they are explicit algorithms for η-expansion and *canonizing substitution*. These principles are discussed in detail in Section 3. Further judgments check that types, contexts, and signatures are well-formed; they are omitted, being entirely straightforward for the propositional fragment.

The types of LLF_0 are freely generated from the constructors $\multimap$, $\rightarrow$, $\&$ and $\top$ and base types. These comprise the largest fragment of intuitionistic linear logic with traditional connectives based on the above judgments for which unique canonical forms exist. This property is essential for the use of LLF_0 as a logical framework, because of the central role of canonical forms in its representation methodology. An alternative characterization is that the fragment consists of all right asynchronous connectives [And92,How98]. The syntax of LLF_0 may be found as a fragment of the syntax for CLF_0 presented in Section 2.2. The typing rules for the canonical variant of LLF_0 are shown in Figure 1.

An example. A Petri net serves as a running example of the various encoding techniques used in these frameworks. The representation of Petri nets in linear logic goes back to Martí-Oliet and Meseguer [MOM91], but has been treated several times in the literature. Familiarity with Petri nets is assumed, and their encoding is

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

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

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

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

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

Fig. 1. Typing rules for LLF$_0$

only given by example. Further details may be found in the companion applications technical report [CPWW02].

Each place in a Petri net is represented by a type constant p. The state of the net is represented as a collection of linear hypotheses: there is an assumption $x{:}p$ for every token in place p. There is also a separate type constant X representing an (unspecific) goal state.

For each transition there is an object constant[4]

$$c_k : (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{X}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{X})$$

expressing that the goal state X can be reached from a state with tokens in places $p_1, \ldots, p_m$ if the goal can be reached from the state with tokens in places $q_1, \ldots, q_n$ instead. Such a rule can be read as removing tokens from $p_1, \ldots, p_m$ and placing them on $q_1, \ldots, q_n$. As an example, consider the Petri net shown in Figure 2 [Cer95].

The initial state of the net is represented by

$$\Delta_0 = r_1 \hat{:} \mathsf{r}, n_1 \hat{:} \mathsf{n}, n_2 \hat{:} \mathsf{n}, b_1 \hat{:} \mathsf{b}, b_2 \hat{:} \mathsf{b}, b_3 \hat{:} \mathsf{b}, a_1 \hat{:} \mathsf{a}$$

and the transitions are represented by the following signature.

$$\begin{array}{ll} \mathsf{P} : (\mathsf{r} \multimap \mathsf{X}) \multimap (\mathsf{p} \multimap \mathsf{X}) & \qquad \mathsf{A} : (\mathsf{c} \multimap \mathsf{X}) \multimap (\mathsf{b} \multimap \mathsf{b} \multimap \mathsf{a} \multimap \mathsf{X}) \\ \mathsf{R} : (\mathsf{p} \multimap \mathsf{n} \multimap \mathsf{b} \multimap \mathsf{X}) \multimap (\mathsf{r} \multimap \mathsf{X}) & \qquad \mathsf{C} : (\mathsf{a} \multimap \mathsf{X}) \multimap (\mathsf{c} \multimap \mathsf{X}) \end{array}$$

The adequacy theorem for this representation states:

Final state $q_1, \ldots, q_n$ can be reached from initial state $p_1, \ldots, p_m$ iff there is a canonical object N such that

$$\cdot; \cdot \vdash N \Leftarrow (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{X}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{X})$$

Moreover, there is a bijection *between sequences of firings of the transition rules of the Petri net and such canonical objects.*

[4] We adopt the convention that the connective $\multimap$ is right associative.

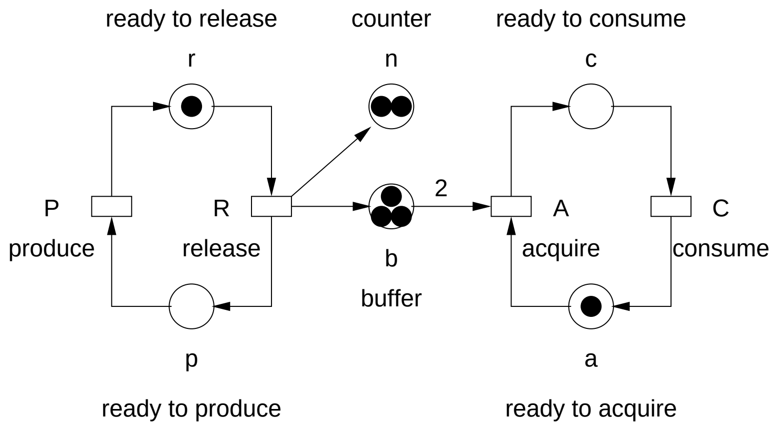

Fig. 2. An example Petri net

Two examples of such objects are as follows. The first represents the firing of R following by the firing of A in the shown initial state. The second shows the same firings in the opposite order. Here the abbreviation $\hat{\lambda} x_1, x_2, \ldots, x_n.\, M$ stands for a curried sequence of linear λ-abstractions. The outermost λ-abstractions have been elided.

$$\begin{array}{l} \cdot; f \hat{:} (\mathsf{c} \multimap \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{p} \multimap \mathsf{X}), \Delta_0 \vdash \\ \qquad \mathsf{R}\hat{}(\hat{\lambda} p_1, n_3, b_4.\, \mathsf{A}\hat{}(\hat{\lambda} c_1.\, f\hat{}c_1\hat{}b_3\hat{}b_4\hat{}n_1\hat{}n_2\hat{}n_3\hat{}p_1)\hat{}b_1\hat{}b_2\hat{}a_1)\hat{}r_1 \Leftarrow \mathsf{X} \\ \cdot; f \hat{:} (\mathsf{c} \multimap \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{p} \multimap \mathsf{X}), \Delta_0 \vdash \\ \qquad \mathsf{A}\hat{}(\hat{\lambda} c_1.\, \mathsf{R}\hat{}(\hat{\lambda} p_1, n_3, b_4.\, f\hat{}c_1\hat{}b_3\hat{}b_4\hat{}n_1\hat{}n_2\hat{}n_3\hat{}p_1)\hat{}r_1)\hat{}b_1\hat{}b_2\hat{}a_1 \Leftarrow \mathsf{X} \end{array}$$

This adequacy theorem captures interleavings of concurrent executions of the net, but not true concurrency. For example, in the initial state of the given net, the R and A transitions can both fire, and do not interfere with each other. We should not be able to distinguish their temporal order; instead, there should be a partial order between possible transitions. The extension of LLF$_0$ to CLF$_0$ is designed to capture this partial order. This issue will be revisited in Section 2.3.

2.2 The Monadic Fragment

It is tempting to think that this issue can be solved by adding more connectives to the framework. Why not work with a framework containing a full complement of linear logical operators (including 1, $A \otimes B$, $!A$) and replace

$$c_k : (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{X}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{X})$$

with the apparently more straightforward

$$c'_k : p_1 \otimes \ldots \otimes p_m \multimap q_1 \otimes \ldots \otimes q_n \ ?$$

The problem is that modeling reachability is not enough; we also want to establish a bijection between Petri-net computations and appropriately typed objects in the framework. If LLF_0 is extended with all connectives of dual intuitionistic linear logic a number of problems establishing adequate encodings arise. The most immediate is that adding an object with a type given by a right synchronous connective can destroy the adequacy of completely unrelated encodings in the framework.

Considering the signature

$$\begin{array}{rclcrcl} \mathsf{nat} & : & \mathsf{type} & \qquad & \mathsf{s} & : & \mathsf{nat} \to \mathsf{nat} \\ \mathsf{z} & : & \mathsf{nat} & & \mathsf{c} & : & 1 \end{array}$$

we see that terms such as ($\mathsf{let}\ 1 = \mathsf{c}\ \mathsf{in}\ \mathsf{z} : \mathsf{nat}$) destroy the bijective correspondence of the type nat with the set of natural numbers.[5] Similar examples would arise in the presence of a constant of type $A \otimes B$, $!A$, $A \oplus B$, or 0. This shows that while a technical conservativity result might hold for LLF_0 and such a language, in practice LLF_0 encodings would be unusable as components of a larger signature including constants having the new types.

More generally, long $\beta\eta$-normal forms either would not exist or would not be unique. At the very least we would be forced to consider a complex set of *commuting conversions*. Their interaction with the needed extensionality principles is poorly understood at present even for the simply-typed λ-calculus with a full set of type constructors, ignoring problems of linearity for the moment. The same is true of proof nets for classical linear logic in the presence of a complete set of connectives [And02].

In order to obtain a tractable, yet sufficiently expressive type theory we employ a technique familiar from functional programming, which does not appear to have been used in logical frameworks or theorem provers: the effects of concurrency are encapsulated in a monad [Mog89]. This encapsulation protects the equational theory of LLF_0. Moreover, the notion of canonical form outside the monad extends the prior notions *conservatively.* This property of the method should not be underestimated, because it means that all encodings already devised for LF or LLF remain adequate, and their adequacy proofs can remain exactly the same!

We write $\{A\}$ for the monad type, which in lax logic would be written $\bigcirc A$ [PD01]. But which types should be available inside the monad? They must be expressive enough to represent the state after a computation step in the concurrent object language. This is most naturally represented by the multiplicative conjunction $\otimes$. Then our transition rule can be written

$$c''_k : p_1 \multimap \ldots \multimap p_m \multimap \{q_1 \otimes \ldots \otimes q_n\}$$

where currying eliminates the use of $\otimes$ on the left-hand side. In order to cover the case $n = 0$ the multiplicative unit 1 is included. Though it does not arise in this example, a transition could also generate an element of persistent (unrestricted) type, so we also allow types $!A$.[6] We call the new types *synchronous*, borrowing terminology from Andreoli [And92], and denote them by S. The full type language is shown in Figure 3.

[5] Examples such as ($\hat{\lambda} x.\ \mathsf{let}\ 1 = x\ \mathsf{in}\ \mathsf{z} : 1 \multimap \mathsf{nat}$) show that the term above cannot simply be equal to z.

[6] A further extension by additive disjunction $\oplus$ and its unit 0 seems plausible but is left to future work.

$$\begin{array}{rcl@{\qquad}rcl}
A, B, C & ::= & a \mid A \multimap B \mid A \rightarrow B & N & ::= & \overset{\wedge}{\lambda} x.\, N \mid \lambda u.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R \\
& & \mid A \,\&\, B \mid \top \mid \{S\} & R & ::= & c \mid u \mid x \mid R^{\wedge} N \mid R\, N \mid \pi_1 R \mid \pi_2 R \\
S & ::= & S_1 \otimes S_2 \mid 1 \mid {!A} \mid A & E & ::= & \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\
& & & M & ::= & M_1 \otimes M_2 \mid 1 \mid {!N} \mid N \\
\Psi & ::= & p \overset{\wedge}{:} S, \Psi \mid \cdot & p & ::= & p_1 \otimes p_2 \mid 1 \mid {!u} \mid x
\end{array}$$

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

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

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

$$\frac{\Gamma, u{:}A; \Delta; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; {!u} \overset{\wedge}{:} {!A}, \Psi \vdash E \leftarrow S}\ !\mathbf{L} \qquad \frac{\Gamma; \Delta, x \overset{\wedge}{:} A; \Psi \vdash E \leftarrow S}{\Gamma; \Delta; x \overset{\wedge}{:} A, \Psi \vdash E \leftarrow S}\ A\mathbf{L}$$

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

Fig. 3. The CLF$_0$ language

The language of objects is extended accordingly. The synchronous types S type *monadic expressions* E. The introduction forms are constructors for multiplicative pairs, unit, and unrestricted canonical objects. The elimination form is a let binding eliminating the monad and matching the synchronous constructors against a pattern p. To our knowledge, this canonical formulation of the proof term assignment for lax logic is novel.

Patterns are classified by synchronous types S, which are collected into a context Ψ, and there are three typing judgments in addition to the judgments already noted for LLF$_0$:

$$\Gamma; \Delta \vdash_\Sigma E \leftarrow S \qquad \Gamma; \Delta; \Psi \vdash_\Sigma E \leftarrow S \qquad \Gamma; \Delta \vdash_\Sigma M \Leftarrow S$$

Additional substitution principles for these judgments are given in Section 3.

The extended language CLF$_0$ inherits all the typing rules already presented for LLF$_0$. The additional typing rules are shown in Figure 3. First, there are introduction and elimination rules for $\{\}$ ($\{\}\mathbf{I}$ $\{\}\mathbf{E}$). We can see that a monadic expression is a sequence of let forms, ending in a monadic object. Immediately after each let the pattern is decomposed into assumptions of the form $x \overset{\wedge}{:} A$ or $u{:}A$ and the body of the let is checked. This is the purpose of the judgment $\Gamma; \Delta; S \vdash E \leftarrow S$, defined by the next group of rules ($\otimes\mathbf{L}$ $1\mathbf{L}$ $!\mathbf{L}$ $A\mathbf{L}$). These correspond to left rules in a sequent calculus. Finally, there are rules to introduce the monadic objects at the end of a sequence of $\{\}\mathbf{E}$ eliminations ($\otimes\mathbf{I}$ $1\mathbf{I}$ $!\mathbf{I}$).

Our example revisited. The Petri net example is now represented almost as in dual intuitionistic linear logic [Bar96], except that the right-hand sides of the linear implications use the monad.

$$\begin{array}{ll} \mathsf{P} : \mathsf{p} \multimap \{\mathsf{r}\} & \mathsf{A} : \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{a} \multimap \{\mathsf{c}\} \\ \mathsf{R} : \mathsf{r} \multimap \{\mathsf{p} \otimes \mathsf{n} \otimes \mathsf{b}\} & \mathsf{C} : \mathsf{c} \multimap \{\mathsf{a}\} \end{array}$$

The monadic encapsulation and the canonical forms of monadic expressions tightly constrain the form of objects constructed from this signature. Adopting α-equivalence (denoted $N_1 \equiv_\alpha N_2$) as the framework's definitional equality, there is an analog of the earlier adequacy theorem.

The example firings are rewritten as follows.

$$\begin{array}{l} \cdot; \Delta_0 \vdash \{\mathsf{let}\ \{p_1 \otimes n_3 \otimes b_4\} = \mathsf{R}\,\hat{}\,r_1\ \mathsf{in}\ \mathsf{let}\ \{c_1\} = \mathsf{A}\,\hat{}\,b_1\,\hat{}\,b_2\,\hat{}\,a_1\ \mathsf{in} \\ \qquad c_1 \otimes b_3 \otimes b_4 \otimes n_1 \otimes n_2 \otimes n_3 \otimes p_1\} \\ \quad \Leftarrow \{\mathsf{c} \otimes \mathsf{b} \otimes \mathsf{b} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{p}\} \\ \cdot; \Delta_0 \vdash \{\mathsf{let}\ \{c_1\} = \mathsf{A}\,\hat{}\,b_1\,\hat{}\,b_2\,\hat{}\,a_1\ \mathsf{in}\ \mathsf{let}\ \{p_1 \otimes n_3 \otimes b_4\} = \mathsf{R}\,\hat{}\,r_1\ \mathsf{in} \\ \qquad c_1 \otimes b_3 \otimes b_4 \otimes n_1 \otimes n_2 \otimes n_3 \otimes p_1\} \\ \quad \Leftarrow \{\mathsf{c} \otimes \mathsf{b} \otimes \mathsf{b} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{p}\} \end{array}$$

Already an advantage over the first LLF$_0$ encoding has been realized in that the spurious goal state X has been eliminated—one might call this a direct-style rather than a continuation-passing-style encoding. But the equality $\equiv_\alpha$ still distinguishes the two executions above despite the fact that their R and A transitions are independent.

2.3 Concurrent Equality

The issue of interleavings of independent computation steps is addressed by a notion of *concurrent equality*, denoted by the following judgments.

$$E \xrightarrow{(R,p)} E' \qquad E_1 \leq E_2 \qquad N_1 \equiv_c N_2 \qquad R_1 \equiv_c R_2 \qquad E_1 \equiv_c E_2 \qquad M_1 \equiv_c M_2$$

Concurrent equality is defined by syntax-directed inference rules on untyped canonical objects. Two expressions E_1 and E_2 should be concurrently equal if each is capable of exhibiting the behavior of the other. In this context "behavior" refers to a sequence of computation steps determined by let forms. The first judgment $E \longrightarrow (R,p)\ E'$ holds when E can perform the computation R, yielding a result matching the pattern p, followed by the additional computation determined by E'. This simple concept is defined by two inference rules.

$$\frac{}{\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \xrightarrow{(R,p)} E} \qquad \frac{E \xrightarrow{(R,p)} E'}{\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E \xrightarrow{(R,p)} \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E'}$$

The second rule is subject to a side condition that the variables bound by the pattern p_1 not be free in the conclusion (so in particular the variables bound by p_1 and p are disjoint). This ensures that only independent computations can be reordered, since the dependence of one computation on another is syntactically evidenced by the occurence of a variable bound by the first computation in the second. Next, the judgment $E_1 \leq E_2$ expresses that E_1 can exhibit the behavior of E_2, characterized as follows.

$$\frac{E_1 \xrightarrow{(R_1,p_2)} E_1' \quad R_1 \equiv_{\mathrm{c}} R_2 \quad E_1' \leq E_2}{E_1 \leq \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2} \qquad \frac{M_1 \equiv_{\mathrm{c}} M_2}{M_1 \leq M_2} \qquad \frac{E_1 \leq E_2 \quad E_2 \leq E_1}{E_1 \equiv_{\mathrm{c}} E_2}$$

Again there is a side condition on the first rule that the variables bound by p_2 not be free in the conclusion. Finally, the concurrent equality itself can be defined. The judgments $N_1 \equiv_{\mathrm{c}} N_2$, $R_1 \equiv_{\mathrm{c}} R_2$ and $M_1 \equiv_{\mathrm{c}} M_2$ are characterized by simple congruence rules for each syntactic form, not shown here. The judgment $E_1 \equiv_{\mathrm{c}} E_2$ holds just when each expression can exhibit the behavior of the other.[7] It is then a simple matter to show that the concurrent equality is an equivalence relation.

Returning to the CLF_0 Petri-net example developed in Section 2.2, it is easy to show that the two objects corresponding to the two different interleavings of the example Petri net execution are concurrently equal. This is crystallized as a better adequacy theorem:

Final state $q_1, \ldots, q_n$ can be reached from initial state $p_1, \ldots, p_m$ iff there is a canonical object N such that

$$\cdot;\cdot \vdash N \Leftarrow p_1 \multimap \ldots \multimap p_m \multimap \{q_1 \otimes \ldots \otimes q_n\}$$

Moreover, there is a bijection *between concurrent executions of the transition rules of the Petri net and equivalence classes of such canonical objects modulo $\equiv_{\mathrm{c}}$.*

While this may seem a minor modification at first, it has far-reaching consequences. Experience with logical frameworks has shown many times that natural encodings lead to deeper understanding of the underlying logical and computational principles, while contrived encodings often do not shed much light on the subject of study. These advantages are multiplied when considering algorithms for manipulating the representations, for proof search, and for meta-theoretic reasoning, because the principles embodied in and provided by the framework have been factored out and do not need to be re-implemented for each encoding. A further discussion of these topics, however, is beyond the scope of the present paper.

3 Identity and Substitution Properties

This section sketches the meta-theory of the canonical formulation of CLF_0. Additional details and a development of the dependent case may be found is the companion theory technical report [WCPW02].

[7] For CLF_0 it can be shown that the judgment $E_1 \leq E_2$ is symmetric, hence one premise of the rule is redundant. However, plausible extensions of the language currently under investigation do not have this property, so we use the more general form given here.

As discussed in Section 2, the CLF$_0$ framework (and full CLF as well) syntactically restrict the form of objects so that they will always be canonical. This means that the identity and substitution principles for typing must be witnessed by transformations on canonical objects.

3.1 The η-Expansion Algorithm

In CLF$_0$ there is no derivation of $\Gamma; x \hat{:} a \to b \vdash x \Leftarrow a \to b$ because the variable rule yields $\Gamma; x \hat{:} a \to b \vdash x \Rightarrow a \to b$ and the coercion rule $\Rightarrow\Leftarrow$ only applies at base type. However, an η-expansion puts the term x into canonical form, and the canonical form does witness the identity principle: $\Gamma; x \hat{:} a \to b \vdash \lambda u.\, x\ u \Leftarrow a \to b$. The notation for general η-expansion is $\eta_A(R) = N$. We have the following identity principle (together with others for the other syntactic categories).

$$\text{If } \Gamma; \Delta \vdash R \Rightarrow A \text{ then } \Gamma; \Delta \vdash \eta_A(R) \Leftarrow A.$$

The η-expansion algorithm is specified by the following equations. Fresh variable names are chosen non-deterministically subject to the condition that new variables introduced on the right-hand side of an equation be distinct from each other and any free variables on the left-hand side. In particular, the notation ν_A generates a fresh pattern of the appropriate form with distinct variables.

$$\begin{array}{l}
\eta_a(R) = R \\
\eta_{A \multimap B}(R) = \hat{\lambda} x.\, \eta_B(R \,{}^\wedge \eta_A(x)) \\
\eta_{A \to B}(R) = \lambda u.\, \eta_B(R\ \eta_A(u)) \\
\eta_{A \& B}(R) = \langle \eta_A(\pi_1 R), \eta_B(\pi_2 R) \rangle \\
\eta_\top(R) = \langle\rangle \\
\eta_{\{S\}}(R) = (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \eta_S(p)) \\
\qquad \text{if } \nu_S = p\ .
\end{array}$$

$$\begin{array}{l}
\nu_{S_1 \otimes S_2} = \nu_{S_1} \otimes \nu_{S_2} \\
\nu_1 = 1 \\
\nu_{!A} = {!u} \\
\nu_A = x \\
\\
\eta_{S_1 \otimes S_2}(p_1 \otimes p_2) = \eta_{S_1}(p_1) \otimes \eta_{S_2}(p_2) \\
\eta_1(1) = 1 \\
\eta_{!A}(!u) = \eta_A(u)
\end{array}$$

3.2 The Canonizing Substitution Algorithm

Similarly, in CLF$_0$ the substitution principle for $\Gamma; \Delta_1 \vdash N_0 \Leftarrow A$ and $\Gamma; \Delta_2, x \hat{:} A \vdash N \Leftarrow B$ relates two different judgments $N_0 \Leftarrow A$ and $x \Rightarrow A$, the latter arising at occurrences of x in the second typing derivation. Since there is no coercion from $\Leftarrow$ to $\Rightarrow$ (which would essentially be a redex), the substitution algorithm must locally reduce introduction forms for the first judgment with elimination forms for the second.

Concretely, substitution is a partial function $[N_0/x{:}A]^{\mathrm{n}} N$. It is characterized by a recurrence, where the recurrence is well-founded for arbitrary (even ill-typed) terms.[8] The well-foundedness of the recurrence is with respect to a lexicographic

[8] This is critical for the staging of the meta-theory in the dependent case [WCPW02].

subexpression order on the triple (A, N_0, N).[9] For substitution, no distinction is made between the two classes of variables u and x; x is used indifferently. We also need the concepts of atomic and principal substitution, discussed below.

The substitution of a canonical 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 elimination forms in the atomic object will be locally reduced with the introduction forms in the canonical object. We refer to this case as *principal substitution.* The result of principal substitution is a canonical object together with its type (used to enforce termination). Principal substitution is denoted $[N_0/x\!:\!A]^\beta R = (N' : A')$. This notation is to be read as a partial function from N_0, A, and R to N' and A'. It will fail to be defined unless x is the head variable of R. The following equations characterize the partial function inductively.

$$
\begin{aligned}
&[N_0/x\!:\!A]^\beta (R\,\hat{}\,N) = ([[N_0/x\!:\!A]^{\mathrm{n}} N/y\!:\!A_1]^{\mathrm{n}} N' : A_2)\\
&\qquad \text{if } [N_0/x\!:\!A]^\beta R = (\hat{\lambda} y.\, N' : A_1 \multimap A_2) \text{ and } A_1 < A\ .\\
&[N_0/x\!:\!A]^\beta (R\ N) = ([[N_0/x\!:\!A]^{\mathrm{n}} N/u\!:\!A_1]^{\mathrm{n}} N' : A_2)\\
&\qquad \text{if } [N_0/x\!:\!A]^\beta R = (\lambda u.\, N' : A_1 \to A_2) \text{ and } A_1 < A\ .\\
&[N_0/x\!:\!A]^\beta (\pi_1 R) = (N_1 : A_1)\\
&\qquad \text{if } [N_0/x\!:\!A]^\beta R = (\langle N_1, N_2\rangle : A_1 \,\&\, A_2)\ .\\
&[N_0/x\!:\!A]^\beta (\pi_2 R) = (N_2 : A_2)\\
&\qquad \text{if } [N_0/x\!:\!A]^\beta R = (\langle N_1, N_2\rangle : A_1 \,\&\, A_2)\ .\\
&[N_0/x\!:\!A]^\beta (x) = (N_0 : A)
\end{aligned}
$$

The threading of the type A through this recurrence is what permits it to be well-founded even for ill-typed terms. The side conditions $A_1 < A$ can be proved redundant once the algorithm has been defined.

Now general substitution into canonical, atomic, and monadic objects can be characterized.

$$
\begin{aligned}
&[N_0/x\!:\!A]^{\mathrm{n}} (\hat{\lambda} y.\, N) = \hat{\lambda} y.\, ([N_0/x\!:\!A]^{\mathrm{n}} N)\\
&[N_0/x\!:\!A]^{\mathrm{n}} (\{E\}) = \{[N_0/x\!:\!A]^{\mathrm{e}} E\}\\
&[N_0/x\!:\!A]^{\mathrm{n}} (R) = [N_0/x\!:\!A]^\beta R \text{ or } [N_0/x\!:\!A]^{\mathrm{r}} R
\end{aligned}
$$

$$
\begin{aligned}
&[N_0/x\!:\!A]^{\mathrm{r}} (c) = c\\
&[N_0/x\!:\!A]^{\mathrm{r}} (y) = y \text{ provided } x \neq y\\
&[N_0/x\!:\!A]^{\mathrm{r}} (R\,\hat{}\,N) = ([N_0/x\!:\!A]^{\mathrm{r}} R)\,\hat{}\,([N_0/x\!:\!A]^{\mathrm{n}} N)
\end{aligned}
$$

$$
[N_0/x\!:\!A]^{\mathrm{m}} (M_1 \otimes M_2) = ([N_0/x\!:\!A]^{\mathrm{m}} M_1) \otimes ([N_0/x\!:\!A]^{\mathrm{m}} M_2)
$$

All these are simple composition laws, except that at the point where an atomic object is coerced to a canonical object, the algorithm non-deterministically chooses

[9] That is, $(A, N_0, N) < (A', N_0', N')$ iff $A < A'$; or $A = A'$ and $N_0 < N_0'$; or $(A, N_0) = (A', N_0')$ and $N < N'$.

between the principal case $[N_0/x{:}A]^\beta R$ and the non-principal case $[N_0/x{:}A]^\mathrm{r} R$. At most one of the two will be defined, depending on whether the head variable of R is or is not x. Cases for some syntactic constructs have been omitted; these are all given by similar composition laws.

It remains to define the substitution into monadic expressions E. Following prior work on proof term assignments for the monadic connectives [PD01], we use "leftist" substitutions which are syntax-directed based on the object at the left of the substitution, rather than the "rightist" substitutions considered above. The recurrences for substitution into expressions and the leftist substitutions are as follows.

$$\begin{array}{l}
\text{if } [N_0/x{:}A]^\beta R = \{E'\} : \{S'\} \text{ and } S' < A \text{ then} \\
\quad [N_0/x{:}A]^\mathrm{e}(\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) = \langle E'/p{:}S'\rangle^\mathrm{e}\, [N_0/x{:}A]^\mathrm{e} E;\ \text{otherwise,} \\
\quad [N_0/x{:}A]^\mathrm{e}(\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) = (\mathsf{let}\ \{p\} = [N_0/x{:}A]^\mathrm{r} R\ \mathsf{in}\ [N_0/x{:}A]^\mathrm{e} E)\ . \\
[N_0/x{:}A]^\mathrm{e}(M) = [N_0/x{:}A]^\mathrm{m} M
\end{array}$$

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

$$\begin{array}{l}
\langle M_1 \otimes M_2/p_1 \otimes p_2{:}S_1 \otimes S_2\rangle^\mathrm{m} E = \langle M_2/p_2{:}S_2\rangle^\mathrm{m}\, \langle M_1/p_1{:}S_1\rangle^\mathrm{m} E \\
\langle 1/1{:}1\rangle^\mathrm{m} E = E \\
\langle !N/!u{:}!A\rangle^\mathrm{m} E = [N/u{:}A]^\mathrm{e} E \\
\langle N/x{:}A\rangle^\mathrm{m} E = [N/x{:}A]^\mathrm{e} E
\end{array}$$

Finally, the substitution theorems can be proved. The following are a few linear cases; there are many more [WCPW02].

1. If $\Gamma; \Delta_1 \vdash N_0 \Leftarrow A$ and $\Gamma; \Delta_2, x\hat{:}A \vdash N \Leftarrow C$ then $\Gamma; \Delta_1, \Delta_2 \vdash [N_0/x{:}A]^\mathrm{n} N \Leftarrow C$
2. If $\Gamma; \Delta_1 \vdash N_0 \Leftarrow A$ and $\Gamma; \Delta_2, x\hat{:}A \vdash R \Rightarrow C$ then $\Gamma; \Delta_1, \Delta_2 \vdash [N_0/x{:}A]^\mathrm{r} R \Rightarrow C$ or $\Gamma; \Delta_1, \Delta_2 \vdash [N_0/x{:}A]^\beta R \Leftarrow C$ as the case may be
3. If $\Gamma; \Delta_1 \vdash E_0 \leftarrow S_0$ and $\Gamma; \Delta_2; p\hat{:}S_0 \vdash E \leftarrow S$ then $\Gamma; \Delta_1, \Delta_2 \vdash \langle E_0/p{:}S_0\rangle^\mathrm{e} E \leftarrow S$

In each case the theorem asserts that the substitution is defined and has the proper type. All these can be proved by a simple structural induction on the well-founded order used to define the algorithm.

Other theorems crucial to the meta-theoretic development include composition laws for η-expansion and substitution, and a functionality principle showing that substitution lifts to a function on equivalence classes of objects modulo $\equiv_\mathrm{c}$ [WCPW02].

4 Related Work

This section presents a brief sketch of related work.

Past research has identified two main approaches to encoding concurrent computations in linear logic. Abramsky's *proofs-as-processes* [BS94] assumes a functional perspective where process interaction is captured by cut-elimination (normalization) steps over linear logic derivations. A second direction, which may be identified with the slogan *proofs-as-traces* (and *formulas-as-processes*), models dynamic process behaviors as proof-search, generally in the style of (linear) logic programming [MOM91,And92,Mil92,KY93,Chi95,Cer95].

CLF follows this second path, stressing a one-to-one correspondence between CLF proof-terms and process executions (traces) [CPWW02]. CLF differs from most of these proposals in two respects: first, it is a fully fledged logical framework, which means that it expresses not only the constructs of an object process calculus and their behavior, but also executions themselves and meta-reasoning about them. Second, the concurrent equality natively supports true concurrency; this is essential for meta-reasoning.

To the authors' knowledge, Honsell et. al. [HMS01] describe the most significant application of a logical framework in the sphere of concurrency. They elegantly encode the π-calculus with substantial meta-theory in the calculus of constructions with inductive/coinductive types ($\mathrm{CC}^{(Co)Ind}$). However, since the notion of equality of $\mathrm{CC}^{(Co)Ind}$ does not identify permutable computations, more advanced meta-theoretic investigations would require tedious coding of an equivalence similar to CLF's concurrent equality.

The idea of monadic encapsulation goes back to Moggi's monadic meta-language [Mog89,Mog91] and is used heavily in functional programming. Our formulation follows the judgmental presentation of Pfenning and Davies [PD01], which completely avoids the need for commuting conversions, but the latter treats neither linearity nor the existence of normal forms. The exploration of monads in logic programming by Bekkers and Tarau [BT95] concentrates on the use of monads for data structures and all-solution predicate. This is quite different from our application and concerned neither with additional logical connectives nor a true extension of the operational semantics. Benton and Wadler [BW96] 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 [dB93] and has been applied to LF by Felty [Fel91]. Our canonical formulation significantly extends and streamlines the ideas behind Felty's *canonical LF* and its extension to LLF [CP98]; the need for confluence and β-normalization results is eliminated.

5 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 [HM94,Cer96]. We conjecture that CLF supports a conservative extension of this operational semantics. We have already constructed a representation of Mini-ML with concurrency and parallelism anticipating such an interpretation [CPWW02].

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.

Case Studies and Applications. Besides the examples presented in this paper, the applications technical report [CPWW02] contains many more examples of the use of the fully dependent framework. We have 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 [Hal85], and concurrency in the style of CML [Rep99]. We further have a representation of the second author's security protocol specification framework MSR [Cer01], and representations of the synchronous and asynchronous π-calculus. Other targets for case studies in the realm of concurrent and imperative languages abound and are left to the reader's imagination.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[And02] Jean-Marc Andreoli. Focussing proof-net construction as a middleware paradigm. In A. Voronkov, editor, *Proceedings of the 18th Conference on Automated Deduction — CADE-18*, pages 501–516, Copenhagen, Denmark, 2002. Springer Verlag LNAI 2392.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[BS94] G. Bellin and P. J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135:11–65, 1994.

[BT95] Yves Bekkers and Paul Tarau. Monadic constructs for logic programming. In J. Lloyd, editor, *Proceedings of the International Logic Programming Symposium (ILPS'95)*, pages 51–65, Portland, Oregon, December 1995. MIT Press.

[BW96] P. N. Benton and Philip Wadler. Linear logic, monads, and the lambda calculus. In E. Clarke, editor, *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 420–431, New Brunswick, New Jersey, July 1996. 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.

[Cer96] Iliano Cervesato. *A Linear Logical Framework.* PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

[Cer01] 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.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, May 1995.

[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.

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Forthcoming.

[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.

[dB93] N.G. de Bruijn. Algorithmic definition of lambda-typed lambda calculus. In G. Huet and G. Plotkin, editors, *Logical Environment*, pages 131–145. Cambridge University Press, 1993.

[Fel91] Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, pages 214–251. Cambridge University Press, 1991.

[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.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)inductive type theories. *Theoretical Computer Science*, 253(2):239–285, 2001.

[How98] Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics.* PhD thesis, University of St. Andrews, Scotland, 1998.

[IP98] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[KY93] Naoki Kobayashi and Akinori Yonezawa. ACL — A concurrent linear logic programming paradigm. In D. Miller, editor, *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294, Vancouver, Canada, 1993. MIT Press.

[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.

[Mog89] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[Pfe01] 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.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[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.