CLF: A logical framework for concurrent systems

Kevin Watkins
Carnegie Mellon University

May 13, 2003

Abstract

The logical framework LF and its instantiation in the Elf language support the specification and implementation of a wide range of logical systems. The concrete representation of object systems of interest is based on the LF methodology of *judgments as types*. Implementations of Elf support checking LF terms, proof search via a logic programming interpretation, representations of meta-theorems and their proofs, and (in the Twelf system) the checking of and automated search for meta-proofs. LF's attractive representation methodology is extended to naturally encompass systems involving state in the linear logical framework LLF, the basic representation principle becoming *state as linear hypotheses*. The LLF representation of stateful systems adequately represents all the possible executions of such systems. However, computations differing only in the order of execution of independent steps are represented by unequal LLF terms. Meta-theoretic investigations including an essential element of concurrency remain cumbersome.

The concurrent logical framework CLF addresses this issue by making concurrency an intrinsic part of the framework's equality. It extends the language of LLF to include all the type constructors of dual intuitionistic linear logic. The additional type constructors $\otimes$, 1, and $\exists$ are associated with let-style elimination forms. The definitional equality of CLF exploits the permutability of these let forms to add an essential element of concurrency. In order to protect the LLF fragment of CLF from the influence of these let forms, a *monadic type constructor* separates the full CLF type language from its LLF fragment. It is of interest in its own right as the first application of monads to the design of a logical framework. The basic representation principle for CLF thus becomes *concurrent computations as monadic expressions*.

The goal of this work is to establish that CLF does indeed enable the succinct and straightforward specification and implementation of concurrent systems. Preliminary work has developed the meta-theory of CLF itself and identified several applications of CLF as a specification language. It remains to develop an implementation of CLF checking, to extend the ideas of Elf to a concurrent logic programming semantics for CLF and implement it, and to consider the meaning of meta-proofs on CLF signatures and how they might be represented.

Thesis committee:	Frank Pfenning, Carnegie Mellon University
	Stephen Brookes, Carnegie Mellon University
	Robert Harper, Carnegie Mellon University
	Gordon Plotkin, University of Edinburgh

1 Introduction

The logical framework LF [HHP93] and its instantiation in the Elf language [Pfe94a] support the specification and implementation of a wide range of logical systems. The concrete representation of object systems of interest is based on the LF methodology of *judgments as types* and *deductions as objects*. Implementations of Elf support checking LF terms, proof search via a logic programming interpretation, representations of meta-theorems and their proofs, and (in the Twelf system [Pfe98]) the checking of and automated search for meta-proofs.

The linear logical framework LLF [Cer96] extends LF's attractive representation methodology to naturally encompass systems involving state, the basic representation principle becoming *state as linear hypotheses*. Processes and computations (or traces) are both represented as LLF objects. In an LLF representation, each possible execution of the system is faithfully represented by an LLF object. However, when a system involves concurrency, computations differing only in the order of execution of independent steps are represented by unequal LLF terms. Compared to the simplicity of the "one deduction, one object" rule for LF representations, this is a serious disadvantage. Meta-theoretic investigations including an essential element of concurrency become cumbersome because the proper notion of equivalence between computations needs to be axiomatized explicitly. The framework cannot offer the same level of support and ease of reasoning for such equivalences that is available for the framework's own intrinsic equality.

The goal of this work is to address the need for a logical framework in the LF tradition that intrinsically models concurrent systems. I propose a concurrent logical framework, CLF, which addresses this problem by introducing an essential element of concurrency into the basic equality of the framework. It extends the language of LLF to include the $\otimes$, 1 and $!$ type constructors of dual intuitionistic linear logic, as well as $\exists$, the dependent cousin of $\otimes$. The additional type constructors are associated with let-style elimination forms. CLF exploits the permutability of these let forms to add an essential element of concurrency. In order to protect the LLF fragment of CLF from the influence of these let forms, a *monadic type* or modality of lax logic separates the full CLF type language from its LLF fragment. The modality is of interest in its own right as the first application of monads to the design of a logical framework. The basic representation principle for CLF thus becomes *concurrent computations as monadic expressions.*

The thesis of this work is that CLF does indeed enable the succinct and straightforward specification and implementation of systems involving an essential ingredient of concurrency. The ultimate goal is to make it as unnecessary to specify and reason explicitly about serializations of a concurrent process as it would be to specify or reason explicitly about matters of α-conversion and capture-avoiding substitution in LF. Another advantage is that it becomes impossible to make certain kinds of mistakes in writing a specification. In LF, for example, it is simply impossible to write a specification that distinguishes α-variants of a term or in which variables are accidentally captured. In CLF it is impossible to write a specification which depends in any way on the order in which independent steps of a concurrent computation are carried out.

Experience has also shown that once a concept is smoothly integrated into the underlying type theory of a framework, the systematization thus achieved makes it much easier to recognize instances of the concept at work in various object systems. This has clearly been the case with "higher-order abstract syntax" [PE88], which has systematized the notion of variable binding and capture-avoiding substitution in a vast range of object languages. Similarly, the linear hypothetical judgment of LLF has been seen to underlie a surprisingly diverse set of phenomena in various object languages. One hopes that CLF's view of concurrent computation could provide a similar unifying principle.

Preliminary work has developed the meta-theory of CLF itself and identified several applications of CLF as a specification language. It remains to develop an implementation of CLF checking, to extend the ideas of Elf to a concurrent logic programming semantics for CLF, and to consider the meaning of meta-proofs on CLF signatures and how they might be represented.

Summary. The remainder of the proposal is organized as follows. Section 2 introduces the CLF type theory, including its syntax, equality judgments and typing judgments. Section 3 discusses how concurrent systems can be represented in CLF and how such representations improve on what is possible in LLF, and relates CLF to various other similar proposals. Section 4 describes CLF's instantiation and expansion operators, which permit the formulation of the framework without a notion of $\beta\eta$-conversion, and describes the key meta-theorems on equality and typing. It also compares the formulation of the type theory seen here with previous accounts of LF and related type theories. Section 5 summarizes what work has been done so far and what is left to do in order to convincingly demonstrate the thesis. Section 6 briefly sets out an agenda for carrying out the remaining tasks. Finally, Section 7 offers some concluding remarks.

2 A type theory with concurrent terms

2.1 A language of concurrent objects

In the LF tradition, the terms classified by types are called *objects*, and the terms classified by kinds are called *type constructors*, among which are the types. CLF has two categories of types: the *asynchronous types* A and the *synchronous types* S. The asynchronous types include all the type constructors of LF and LLF, as well as a new *monadic type constructor* written $\{S\}$. The synchronous types, which are only allowed within the monad constructor, include further type constructors of intuitionistic linear logic, all of which have let-style elimination rules. Intuitively, the monad restricts the availability of these elimination forms so that the question of commutative conversions between the eliminations and other terms of the type theory does not arise. Sections 2.3 and 3.3 describe this in further detail. The "(a)synchronous" terminology is due to Andreoli [And92].

Definition 1 (Type constructors)

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

The asynchronous types include the dependent function type $\Pi x{:}A.\,B$, the linear function type $A \multimap B$, the additive product type $A \,\&\, B$, and the additive unit $\top$. The monadic type $\{S\}$ acts as a coercion from the synchronous types into the asynchronous types. The *atomic type constructors* P include type constructor constants a and the type-level dependent application $P\ N$ (where N is an object).

The synchronous types include the other type constructors new to CLF: the multiplicative product type $S_1 \otimes S_2$, the multiplicative unit 1, and the dependent pair type $\exists x\!:\!A.\,S$. There is a trivial coercion from the asynchronous types into the synchronous ones. In addition, the exponential type ! of intuitionistic linear logic can be defined as a trivial dependent pair: $!A \equiv \exists x\!:\!A.\,1$.

The kind language of CLF is identical to that of LF and LLF. The symbol kind is also used on occasion to classify the valid kinds.

Definition 2 (Kinds)

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

We often write $A \to B$ instead of $\Pi x\!:\!A.\,B$ and $A \to K$ instead of $\Pi x\!:\!A.\,K$ when B or K respectively contains no free occurrence of x.

The CLF type theory inherits all the type and kind constructors of LF and LLF, and the corresponding objects. It differs, however, in that the syntax of CLF admits only those terms of LF and LLF that are β-normal and η-long—the *canonical* terms. This simplifies the meta-theory of CLF, and highlights the importance of the principle that LF representations always establish a compositional bijection between terms of an object language and the canonical objects of LF of a given type. Thus, in CLF, there is no notion of β- or η-conversion, and every well-formed term can be regarded as "canonical."

Definition 3 (Objects)

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

The first two categories of object are the *normal objects* N and the *atomic objects* R. These correspond to the quasi-canonical and quasi-atomic forms of LF object, respectively, as described by Harper and Pfenning [HP00]. A normal object is a series of constructors applied to atomic objects, while an atomic object is a series of natural-deduction style destructors applied to a variable x or constant c. They include all the constructors and destructors of LF and LLF: the unrestricted function constructor $\lambda x.\,N$ and destructor $R\;N$; the linear function constructor $\hat{\lambda} x.\,N$ and destructor $R\,\hat{}\,N$; the additive pair constructor $\langle N_1, N_2\rangle$ and destructors $\pi_1 R$ and $\pi_2 R$; and the additive unit constructor $\langle\rangle$.

In addition, there is a constructor $\{E\}$ associated with the monadic type. The remaining categories of object, the *expressions* E and *monadic objects* M, are associated with the monadic type and the additional linear type constructors $\otimes$, 1 and $\exists$. They include the monadic binding form $\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E$; the multiplicative pair constructor $M_1 \otimes M_2$ and pattern $p_1 \otimes p_2$; the multiplicative unit constructor 1 and pattern 1; and the dependent pair constructor $[N, M]$ and pattern $[x, p]$. The monadic binding form $\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E$ binds all the variables occurring in the pattern p within the expression E. Hence, it subsumes the destructors for 1, $\otimes$, and $\exists$. The purpose of the monadic type is to isolate these binding forms, which would otherwise have a catastrophic effect on the LF and LLF fragments of CLF, as explained in Section 3.2.

Terms which differ only in the names of their bound variables are considered to be the same. For the LF and LLF fragments of CLF, this is the only notion of equality: two terms are equal if and only if they are α-equivalent. But expressions differ from the other categories of object in that they are subject to *permutative conversions* by which the monadic bindings can be reordered:

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

Of course, this rule is subject to the proviso that the bindings be *independent*: p_1 and p_2 must bind disjoint sets of variables, no variable bound by p_1 can appear free in R_2, and vice versa. The reordering of monadic bindings is the mechanism by which CLF admits an intrinsic description of concurrent computations. We think of each let binding as a single computation step. Computation steps appearing in a single expression that are independent in the above sense can be thought of as occurring concurrently.

The notion of equality on CLF objects could be characterized as the least congruence relation including the above equation schema. The reason is that having separate syntactic classes of objects and expressions eliminates any need for *commuting conversions*. (We do not think of the permutative conversions as being commuting conversions.) But we prefer to define the framework's equality in a slightly different way. The definition relies on the subsidiary concept of a *concurrent context*.

Definition 4 (Concurrent contexts)

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

As usual, the notation $\epsilon[E]$ stands for the expression constructed by replacing the hole $\_$ in ϵ with E. Now equality can be defined as follows.

Definition 5 (Equality)

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

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

$E_1 = E_2$ [Expression equality]

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

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

(All congruences.)

The judgment $E_1 =_{\mathrm{c}} E_2$ holds when E_1 and E_2 represent the same underlying concurrent computation even though their syntactic representations may differ. The rule marked (*) is subject to the side condition that no variable bound by p be free in the conclusion or bound by the context ϵ, and that no variable free in R_2 be bound by the context ϵ. (It is always possible to globally α-convert terms being compared for equality in order to avoid running afoul of the side condition.)

Then the equality on expressions $E_1 = E_2$ is simply defined to be concurrent equality. While the definition of concurrent equality is not presented symmetrically, it turns out to be symmetric. The judgments $N_1 = N_2$, $R_1 = R_2$, $M_1 = M_2$, and $P_1 = P_2$ are characterized by simple congruence rules for each syntactic form, not shown. In Section 4.5 it is shown that equality is an equivalence relation. When it is necessary to refer to α-equivalence, as opposed to the framework's equality, the former will be denoted $\equiv$.

The main advantage of this concrete definition of equality is that it is syntax directed. The sequence of rules to be applied is determined by the syntax of the term on the left, and the instantiation of the metavariables in the rule schemas is determined up to finitely many possibilities (in considering how to decompose an expression E on the right into $\epsilon[E']$). Hence it is manifestly decidable and lends itself to interpretation as an algorithm.

2.2 The type system of CLF

We present first the type system of the LF fragment of CLF, then extend it to the LLF fragment and finally the full language. Before presenting the typing judgments, it is necessary to introduce the notions of *signature* and *context*, which record assumptions about the types (or kinds) of constants and variables respectively.

Definition 6 (Signatures and contexts, LF fragment)

$$\begin{array}{ll} \Sigma ::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A & \textit{Signatures} \\ \Gamma ::= \cdot \mid \Gamma, x{:}A & \textit{Unrestricted contexts} \end{array}$$

It is an invariant that variables added to signatures and contexts must be unique, and the metavariables for contexts always denote contexts with unique variables.

There is a typing judgment for each syntactic category, as well as validity judgments for contexts Γ and signatures Σ. Each of these judgments is defined in a completely syntax-directed manner, so termination and decidability of typing is clear. For *normal* syntactic categories (N, A, K) the operational interpretation of the type-checking judgment is that a putative type is provided, and the judgment holds if the term can be typed with the given type. In particular, a normal term such as $\lambda x.\, x$ may have several different types. This stands in contrast to the typical presentation of LF, where type labels are used in abstractions to ensure that every term has a unique type. For the *atomic* syntactic categories (R, P) the situation is different: the operational meaning of the typing judgment is that it defines a partial function from an atomic term (in a given context and signature) to its unique type. The direction of the arrow $\Leftarrow$ or $\Rightarrow$ serves as a mnemonic for whether a type is being checked against or inferred, respectively.

In all cases the typing judgment is not considered to have any particular meaning unless the context and signature referred to in the judgment are valid. For the normal syntactic categories, the typing judgment is meaningless unless the type referred to in the judgment is valid as well. For the atomic syntactic categories, it will be proved that whenever a typing is derivable and the context and signature mentioned in the typing are valid, the type mentioned in the judgment is valid. The signature Σ subscripting the various judgments is often omitted—it is invariant in the course of a typing derivation.

Definition 7 (Typing, LF fragment)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The typing rules [Π**KE**] and [Π**E**] involve the operation of instantiating a variable in a dependent type (or kind) with an object. In the first-order case, an ordinary capture-avoiding substitution will suffice. However, since β-redices are not syntactically allowed in CLF, at higher type some computation must occur in order to find the term corresponding to the result of the instantiation. This is achieved by the *instantiation operator* $\mathsf{inst\_a}_A(x.\, B, N)$, which computes the result of instantiating the variable x in the type B with the object N. The instantiation operator is indexed by the type A of the object being substituted. In the first-order case, we have that $\mathsf{inst\_a}_{\mathsf{a}}(x.\, B, N) \equiv [N/x]B$, but at higher type, more complex situations arise:

$$\mathsf{inst\_a}_{\mathsf{a}\to\mathsf{a}}(x.\, \mathsf{b}\ (\lambda y.\, \mathsf{c}\ (x\ (x\ y))),\ \lambda z.\, \mathsf{d}\ z) \equiv \mathsf{b}\ (\lambda y.\, \mathsf{c}\ (\mathsf{d}\ (\mathsf{d}\ y)))$$

The instantiation operators for each syntactic category are defined in Section 4.1.

In contrast to the usual presentation of LF typing, there are no type conversion rules (which would fail to be syntax directed). Instead, there is an appeal to the equality judgment in the rule [$\Rightarrow\Leftarrow$] for the coercion from atomic objects to normal objects. In Section 4.5 type conversion for the typing judgments for normal syntactic categories is shown to be admissible.

In order to extend the type system to the LLF fragment, we introduce a new context for linear hypotheses. We depart slightly from the concrete syntax in adhering to the usual convention that a metavariable like Δ denotes an equivalence class of linear contexts up to rearrangement. Note that there is no issue of dependency, as there would be in rearranging an unrestricted context Γ, because types cannot depend on linear variables.

Definition 8 (Contexts, LLF fragment)

$$\Delta ::= \cdot \mid \Delta, x \hat{:} A \qquad \textit{Linear contexts}$$

The typing judgments for objects must be modified in order to account for linear hypotheses—the new formulation depends on a pair of contexts $\Gamma; \Delta$ in the style of dual intuitionistic linear logic [Bar96]. The following definition includes all the inference rules from the LF fragment that have to be revised for this reason. Note that type constructors and kinds never depend on linear variables.

Definition 9 (Typing, LLF fragment)

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

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

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

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

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

$\Gamma \vdash_\Sigma P \Rightarrow K$ [Atomic type inference, revised]

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

$\Gamma; \Delta \vdash_\Sigma N \Leftarrow A$ [Normal object checking, revised and extended]

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

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

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

$\Gamma; \Delta \vdash_\Sigma R \Rightarrow A$ [Atomic object inference, revised and extended]

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

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

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

Finally, we are ready to extend the type system to the full CLF language. This requires one more kind of context to record the types of patterns.

DEFINITION 10 (CONTEXTS, FULL CLF)

$$\Psi ::= \cdot \mid p \mathbin{\hat{:}} S, \Psi \qquad \textit{Pattern contexts}$$

There are additional judgments for the validity of pattern contexts and synchronous types, and for typing expressions and monadic objects. Note that pattern contexts Ψ are ordered but do not allow dependencies.

DEFINITION 11 (TYPING, FULL CLF)

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

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

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

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

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

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

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

$\Gamma; \Delta \vdash_\Sigma N \Leftarrow A$ [Normal object checking, extended]

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

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

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

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

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

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

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

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

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

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

The rule [$\exists$**I**] requires another kind of instantiation operator in order to instantiate the dependent variable in a dependent pair type.

2.3 Related work

As noted above, CLF includes the LF and LLF frameworks as fragments, where the CLF counterpart of an LF or LLF object is its canonical form with type labels omitted. The CLF extension is conservative; anyone who knows how to use LF or LLF can bring their (canonical) specifications to CLF and use them without modification. There is a further "modularity" property: given an LF or LLF signature Σ_1 and a disjoint CLF signature Σ_2, both of which are valid, and a type A well-formed in Σ_1, the set of objects of type A in Σ_1, Σ_2 is the same as the set of objects of type A in Σ_1 alone.[1]

The LLF framework was motivated as the largest fragment of intuitionistic linear logic having a proof term assignment without commuting conversions [Cer96]. The equational theory that would be associated with commuting conversions was seen as intractable. Pfenning and Davies note that the commuting conversions of Moggi's monadic metalanguage [Mog89, Mog91] can be eliminated by creating a new typing judgment associated with a new class of object, the expressions [PD01]. They exhibit a compositional translation from the monadic metalanguage into their proof term assignment for lax logic [FM97]. The translation takes monadic metalanguage terms related by commuting conversions into identical terms of the target language. CLF exploits their idea to eliminate commuting conversions that would otherwise be associated with the monadic type and the synchronous types. The monadic type constructor $\{S\}$ would be written $\bigcirc S$ in lax logic.

Further remarks on how CLF relates to frameworks outside the LF family appear at the end of the following section.

3 Toward a methodology for representing concurrent systems

A logical framework in the LF tradition is not only a type theory. It is also a methodology for representing the deductive systems of interest within that type theory [Pfe99]. The LF methodology represents object-language judgments by LF types and object-language deductions by LF objects. Other syntactic entities of the object language (propositions, expressions, types, etc.) are also represented by LF objects. In each case an *adequacy theorem* establishes that there is a compositional bijection between well-formed entities of the object language and well-typed canonical objects in LF (of a certain type). The higher types and dependent types available in LF assist in this task: LF abstraction models object-language binding, α-conversion, and substitution ("higher-order abstract syntax" [PE88]); LF abstraction also models hypothetical and parametric judgments of the object language; and LF dependent function types enforce well-formedness constraints on object-language deductions.

For this reason, LF representations do not need to deal explicitly with matters of variable binding and α-conversion, and certain kinds of errors in specification associated with such binding constructs are impossible to make. Similarly, LF representations do not need to explicitly encode the relation between an object-language deduction and the proposition that it proves—the proposition that a deduction proves becomes an intrinsic part of its

[1]The modularity result does not hold for arbitrary disjoint CLF signatures Σ_1 and Σ_2. It could be recovered by replacing the monadic type constructor with a countably infinite family of "tagged" monadic type constructors and requiring that the sets of tags mentioned in Σ_1 and Σ_2 be disjoint.

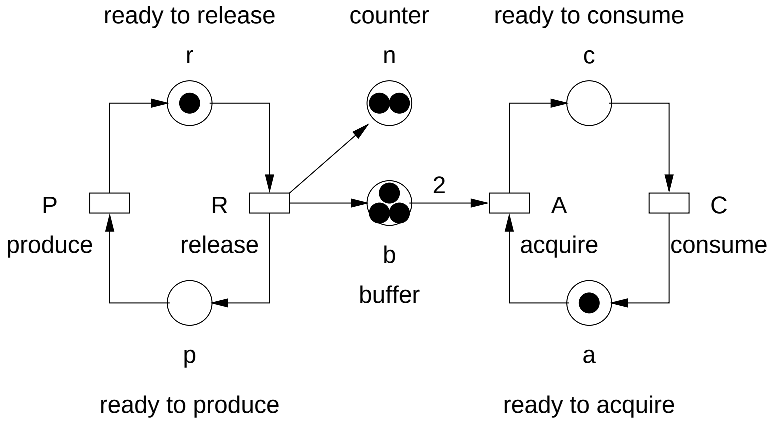

Figure 1: An example Petri net

(dependent) LF type. In this way, proof checking reduces to LF type checking, which is decidable and efficient.

Linear LF seeks to expand these benefits to the case of *linear* hypothetical judgments. Linear hypotheses make *mutable state* an intrinsic part of the framework. When representing an object system involving mutable state, it is not necessary to represent the state as an explicit object, nor to explicitly specify operations for adding information to the state, changing it, or withdrawing information from it. This idea takes on a particularly simple form when the state consists of the presence or absence of any of a set of discrete *resources*, as in the following example.

3.1 An example: Petri nets with labeled tokens

Here we consider a simple example of an object system involving state: a Petri net [Pet62] with labeled tokens. Such a Petri net consists of a directed bipartite graph of *places* and *transitions*. A state of the net consists of a mapping from a finite set of *tokens*, each uniquely labeled, to the places.[2] A computation step consists of choosing a transition, removing a token from each of its antecedent places, and adding a fresh token to each of its succedent places. A transition cannot be chosen if any of its antecedent places has no tokens.

For example, Figure 1 shows a labeled Petri net with six places and four transitions. Our simple representation of the net in the LLF fragment of CLF models the places by LLF type constants and the transitions and tokens by LLF objects. Each token has a type corresponding to the place in which it is located. (A more sophisticated representation might model places as objects and introduce a two-place judgment "token x is in place y.") The transitions are linear functions in continuation-passing style, "consuming" linear hypotheses associated with the antecedent places and introducing new linear hypotheses associated with the succedent places. A type G represents the continuation. Thus we have the following signature:

[2]One requirement of the LLF representation methodology, when applied to systems involving discrete sets of resources, is that the resources be *distinguishable*. Hence the tokens of the Petri net must carry unique labels. It is possible that proof irrelevance [Pfe01] could offer a way of modeling indistinguishability.

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

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 an object N such that*
>
> $$\cdot; \cdot \vdash N \Leftarrow (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{G}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{G})$$
>
> *Moreover, there is a* bijection *between sequences of firings of the transition rules of the Petri net and such canonical objects.*

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 $\hat{\lambda} x_1, x_2, \ldots, x_n. N$ stands for a curried sequence of linear λ-abstractions. The outermost λ-abstractions have been elided.

$$
\begin{array}{rl}
\Delta \;=\; & f \mathbin{\hat{:}} (\mathsf{c} \multimap \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{p} \multimap \mathsf{G}), \\
& \qquad\qquad r_1 \mathbin{\hat{:}} \mathsf{r}, n_1 \mathbin{\hat{:}} \mathsf{n}, n_2 \mathbin{\hat{:}} \mathsf{n}, b_1 \mathbin{\hat{:}} \mathsf{b}, b_2 \mathbin{\hat{:}} \mathsf{b}, b_3 \mathbin{\hat{:}} \mathsf{b}, a_1 \mathbin{\hat{:}} \mathsf{a}
\end{array}
$$

$$
\begin{array}{l}
\cdot; \Delta \vdash \mathsf{R}\,{}^\wedge(\hat{\lambda} p_1, n_3, b_4.\, \mathsf{A}\,{}^\wedge(\hat{\lambda} c_1.\, f\,{}^\wedge c_1\,{}^\wedge b_3\,{}^\wedge b_4\,{}^\wedge n_1\,{}^\wedge n_2\,{}^\wedge n_3\,{}^\wedge p_1)\,{}^\wedge b_1\,{}^\wedge b_2\,{}^\wedge a_1)\,{}^\wedge r_1 \Leftarrow \mathsf{G} \\
\cdot; \Delta \vdash \mathsf{A}\,{}^\wedge(\hat{\lambda} c_1.\, \mathsf{R}\,{}^\wedge(\hat{\lambda} p_1, n_3, b_4.\, f\,{}^\wedge c_1\,{}^\wedge b_3\,{}^\wedge b_4\,{}^\wedge n_1\,{}^\wedge n_2\,{}^\wedge n_3\,{}^\wedge p_1)\,{}^\wedge r_1)\,{}^\wedge b_1\,{}^\wedge b_2\,{}^\wedge a_1 \Leftarrow \mathsf{G}
\end{array}
$$

The benefit of the LLF methodology is clear: it is not necessary to explicitly manage a "list" of tokens and axiomatize operations for adding and removing tokens from the list, as one would have to do in LF. In LF one would also have to prove various interaction laws for the token-list-managing operations; in LLF, the required principles are proved once and for all as structural laws of the framework's linear hypothetical judgment. However, there is also room for improvement. We might have hoped for an adequacy theorem relating LLF objects to *concurrent computations* of the Petri net, that is, equivalence classes of computations under rearragement of independent steps. But this strengthened adequacy theorem does not hold. For example, the two LLF terms above correspond to computations differing only in the order of independent R and A steps: no labeled token indexing the R step is involved in the A step, and vice versa. But the structure of the LLF representation nonetheless requires that the two orderings be represented by different terms. In essence, the continuation-passing style of the representation forces a sequentialization of the computation.

3.2 Representing LPNs in CLF

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

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

with the apparently more straightforward

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

Unfortunately, such an extension would not be conservative over the LF and LLF fragments of the type language. In fact, it would have a catastrophic effect on adequacy results for even very simple LF encodings. Consider a simple LF representation of the natural numbers together with an additional constant c of multiplicative unit type:

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

The problem is 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.[3] Similar examples would arise in the presence of a constant of type $A \otimes B$, $!A$, $A \oplus B$, or 0. So the adequacy of the LF encoding is destroyed by the presence of even a single object constant having a type given by one of the new type constructors.

The underlying problem is that the destructors associated with these *synchronous* types involve "polymorphic" binding constructs that do not constrain the type of the object resulting from the binding. CLF's monadic type extracts us from this difficulty by restricting the awkward binding constructs to the monad. This encapsulation protects the pure LF and LLF fragments of CLF from the new constructs. All encodings already devised for LF or LLF remain adequate, and their adequacy proofs can remain exactly the same. Furthermore, as has already been noted, the separation between expressions and object rules out commuting conversions, simplifying the equational theory.

Using CLF's multiplicative conjunction $\otimes$, our transition rule can be rewritten as

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

where currying eliminates the use of $\otimes$ on the left-hand side. The multiplicative unit 1 covers the case $n = 0$. Though it does not arise here, a modified model of Petri nets might also have transitions generating elements of persistent (unrestricted) type. In that case we would have exponentials $!q$ on the right as well. (Exponentials on the left can be curried away using the unrestricted function type.)

As an aside, while one might think that the presence of the exponential type constructor ! would render the intuitionistic function type $A \to B$ definable by $\{!A\} \multimap B$, this is not in fact the case. As a simple counterexample we may see that while $a \to a$ (a being an uninhabited atomic type) is inhabited by $\lambda x.\, x$, the type $\{!a\} \multimap a$ is not inhabited. The type $\{!a\} \multimap \{a\}$ is inhabited by $\hat{\lambda} x.\, \{\mathsf{let}\ \{!y\} = x\ \mathsf{in}\ y\}$, however.

Thus, in CLF, the Petri net example is represented almost as in intuitionistic linear logic, except that the right-hand sides of the linear implications use the monad.

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

The two example Petri net executions shown above in the LLF encoding correspond to the following objects in the new encoding:

[3] 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.

$$\begin{array}{rcl} \Delta & = & r_1 \hat{:} r, n_1 \hat{:} n, n_2 \hat{:} n, b_1 \hat{:} b, b_2 \hat{:} b, b_3 \hat{:} b, a_1 \hat{:} a \\ S & = & c \otimes b \otimes b \otimes n \otimes n \otimes n \otimes p \end{array}$$

$$\begin{array}{l} \cdot; \Delta \vdash \{\mathsf{let}\ \{p_1 \otimes n_3 \otimes b_4\} = R\,^\wedge r_1\ \mathsf{in}\ \mathsf{let}\ \{c_1\} = A\,^\wedge b_1\,^\wedge b_2\,^\wedge a_1\ \mathsf{in} \\ \qquad\qquad c_1 \otimes b_3 \otimes b_4 \otimes n_1 \otimes n_2 \otimes n_3 \otimes p_1\} \Leftarrow \{S\} \\ \cdot; \Delta \vdash \{\mathsf{let}\ \{c_1\} = A\,^\wedge b_1\,^\wedge b_2\,^\wedge a_1\ \mathsf{in}\ \mathsf{let}\ \{p_1 \otimes n_3 \otimes b_4\} = R\,^\wedge r_1\ \mathsf{in} \\ \qquad\qquad c_1 \otimes b_3 \otimes b_4 \otimes n_1 \otimes n_2 \otimes n_3 \otimes p_1\} \Leftarrow \{S\} \end{array}$$

It is now easy to see that the two executions are equal. This idea is crystallized as an improved adequacy theorem:

> *Final state $q_1, \ldots, q_n$ can be reached from initial state $p_1, \ldots, p_m$ iff there is a 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 objects modulo $=$.*

While at first this may seem a minor modification, 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.

The representation principle for CLF, then, can be summarized by "concurrent computations as monadic expressions." Each computation step in a concurrent computation becomes a binding $\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E$ possibly "consuming" some linear hypotheses in R and "producing" more linear hypotheses in p, which will be available in the rest of the computation E.

While there is not space here to discuss more interesting examples of CLF representations, Appendix A shows an example concerning the synchronous π-calculus [Mil99].

3.3 Related work

The LF representation methodology is discussed in detail in the handbook article on logical frameworks by Pfenning [Pfe99]. Further examples of CLF representations, including a full development of the Petri net example, are in the technical report by Cervesato et al. [CPWW02]. The idea of representing Petri nets by linear hypothetical judgments has a long history, going back to Martí-Oliet and Meseguer [MOM91].

Monadic encapsulation in the context of functional programming and type theory begins with Moggi's monadic metalanguage [Mog89, Mog91]. Prawitz describes a proof theory for modal logics [Pra65], from which Moggi's presentation of the monadic metalanguage inherits. Pfenning and Davies [PD01] revisit the question of proof theory for modal logic, reinterpreting it on the basis of a judgmental approach in the style promulgated by Martin-Löf [ML96]. Their approach improves on the original formulation of the monadic metalanguage in that commuting conversions are no longer needed. Their judgmental analysis of the modal operator of lax logic is the basis for CLF's modal type constructor and the associated CLF judgments. They do not, however, consider the existence of canonical forms, and their proof term language, unlike CLF, relies on $\beta\eta$-conversion.

There have been many other formalisms proposed for the representation of concurrent systems, many having elements in common with the CLF approach. Abramsky's "proofs-as-processes" relates classical linear logic with the synchronous π-calculus [Abr93, BS94]. Here concurrent computation corresponds to proof normalization (cut elimination), giving the system a functional flavor. Concurrent computations (traces) are thus not reified as objects, as they are in CLF.

Closer to the CLF view are approaches in which logical formulas are identified with processes and proofs with concurrent computations. Thus, these are nearer to logic programming in the sense of proof search [MNPS91] than to functional programming. For example, Miller outlines a translation from the π-calculus into linear logic: processes become LL propositions and π-calculus reduction becomes LL entailment [Mil92]. These ideas are generalized and reformulated as a logical framework in Miller's proposal for the specification logic Forum [Mil96a, Chi95]. Forum offers a paradigm for viewing logic programming as concurrent computation, and so it is likely that the logic programming interpretation for CLF will draw heavily from it. However, in Forum proofs cannot be manipulated as first-class objects—not even cut elimination is treated, let alone an equational theory on proofs. The same is true of LinLog [And92], LO [AP91], ACL [KY93], Lygon [HPW96], and LLP [HWTK98], all of which treat logic programming over other fragments of classical or intuitionistic linear logic.

Perrier describes how the basic idea of linear logic programming as concurrent computation can be improved by adopting *proof nets* [Gir87] rather than sequent calculus proofs as the fundamental computational objects [Per98]. Again, however, proof nets are treated only meta-theoretically—they are not first-class terms of the process language. The key foundational idea of Elf, that proofs should be just like any other objects of the framework, does not apply.

Barber proposes a very general type theory based on linear operators and proves decidability in the general framework [Bar97]. Since the generalization includes the DILL type theory and the action calculus as special cases, this immediately entails the decidability of the DILL type theory. However, there is no treatment of dependent types, nor would DILL be adequate for LF-style representations for the reasons outlined in Section 3.2. Also, the presentation does not seem to lend itself to concrete implementation as a mechanized framework. Finally, Barber's reliance on proof nets would seem to render the extension to additive sum types problematic, while for CLF it is straightforward, because the issue of commuting conversions (other than an appropriate generalization of the permutative conversions) does not arise.

Honsell et al. develop perhaps the most significant application of a logical framework in the sphere of concurrency [HMS01]. They present an encoding of the π-calculus in the Coq system, a framework based on a higher-order type theory with (co)inductive types, and work out some rather advanced meta-theory using the encoding. But this should be regarded as a *tour de force*, given that Coq offers no intrinsic support for reasoning about concurrency. Of course, even in CLF, abstract relations like the *strong late bisimularity* treated in their development would need to be treated explicitly. CLF's concurrent equality simplifies such reasoning; it does not obviate the need for it.

4 Some meta-theoretic results

The design of CLF is based on the idea that every syntactically well-formed term should be "canonical," and that the notion of $\beta\eta$-conversion should be eliminated, in favor of simple inductively defined *instantiation operators* for instantiating a variable in a term with an object. Thus, in developing the meta-theory of CLF there is no need to consider issues such as confluence and normalizability *per se*. Instead, we define the instantiation operators by a manifestly terminating recurrence, and we focus on the simple algebraic laws that the instantiation operators satisfy, and on their interaction with the framework's notion of equality.

Another new element of this approach is an *expansion operator* taking an atomic object to the corresponding normal object at higher type. This is necessary because the coercion rule $[\Rightarrow\Leftarrow]$ from atomic objects to normal objects can only be applied at base type. The expansion operator thus replaces the idea of η-conversion.

Once the instantiation and expansion operators have been defined, and their algebraic properties characterized, it is possible to describe their relationship to the judgments for typing. It so happens that the instantiation operators witness the familiar substitution principles for typing (the transitivity of entailment, from the proof-theoretic point of view), and the expansion operator witnesses the identity principles for typing (the reflexivity of entailment).

4.1 Instantiation

The recurrence defining instantiation is based on the observation, exploited in cut elimination proofs on the logical side [Pfe00b], but not so well known on the type theoretic side, that the canonical result of substituting one canonical term into another can be defined by induction on the type of the term being substituted. Accordingly, the instantiation operators are defined as a family parameterized over the type of the object being substituted. In the notation $\mathsf{inst\_}c_A(x.\,X, N)$ this type A appears as a subscript. Here c is replaced by a mnemonic for the particular syntactic category to which the instantiation operator applies. The variable x is to be considered bound within the term X (of whatever category) being substituted into. The operators defined in this section should be thought of as applying to equivalence classes of concrete terms modulo α-equivalence on bound variables.

Together with the instantiation operators, and defined by mutual recursion with them, is a *reduction operator* $\mathsf{reduce}_A(x.\,R, N)$ that computes the canonical object resulting from the instantiation of x with N in the case that the *head variable* $\mathrm{head}(R)$ of the atomic object R is x. Thus, roughly speaking, it corresponds to the idea of weak head reduction for systems with β-reduction. The instantiation operator $\mathsf{inst\_r}_A(x.\,R, N)$, by contrast, is only defined if the head of R is *not* x. Another distinguishing feature is that reduction on an atomic object yields a normal object, while instantiation on an atomic object yields an atomic object.

Finally, there is a *type reduction operator* $\mathsf{treduce}_A(x.\,R)$ that computes the putative type of R given that the head of R is x and the type of x is A.[4] Type reduction is used in side conditions that ensure that the recurrence defining instantiation is well-founded.

The first definition covers the LF fragment of CLF.

[4] Actually, to be more precise, the type of R will be a substitution instance of $\mathsf{treduce}_A(x.\,R)$. The instantiation operators do not keep track of dependencies within the type subscript.

Definition 12 (Instantiation, LF fragment)

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

$$\mathsf{treduce}_A(x.\,x) \equiv A$$
$$\mathsf{treduce}_A(x.\,R\;N) \equiv C \quad \text{if } \mathsf{treduce}_A(x.\,R) \equiv \Pi y\!:\!B.\,C$$

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

$$\mathsf{reduce}_A(x.\,x, N_0) \equiv N_0$$
$$\mathsf{reduce}_A(x.\,R\;N, N_0) \equiv \mathsf{inst\_n}_B(y.\,N', \mathsf{inst\_n}_A(x.\,N, N_0))$$
$$\quad \text{if } \mathsf{treduce}_A(x.\,R) \equiv \Pi y\!:\!B.\,C \text{ and } \mathsf{reduce}_A(x.\,R, N_0) \equiv \lambda y.\,N'$$

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

$$\mathsf{inst\_r}_A(x.\,c, N_0) \equiv c$$
$$\mathsf{inst\_r}_A(x.\,y, N_0) \equiv y \quad \text{if } y \text{ is not } x$$
$$\mathsf{inst\_r}_A(x.\,R\;N, N_0) \equiv (\mathsf{inst\_r}_A(x.\,R, N_0))\;(\mathsf{inst\_n}_A(x.\,N, N_0))$$

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

$$\mathsf{inst\_n}_A(x.\,\lambda y.\,N, N_0) \equiv \lambda y.\,\mathsf{inst\_n}_A(x.\,N, N_0) \quad \text{if } y \notin \mathrm{FV}(N_0)$$
$$\mathsf{inst\_n}_A(x.\,R, N_0) \equiv \mathsf{inst\_r}_A(x.\,R, N_0) \quad \text{if head}(R) \text{ is not } x$$
$$\mathsf{inst\_n}_A(x.\,R, N_0) \equiv \mathsf{reduce}_A(x.\,R, N_0) \quad \text{if } \mathsf{treduce}_A(x.\,R) \equiv P$$

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

(Analogous.)

The recurrence defining these operators is based on a structural induction. There is an outer induction on the type subscripting the operators, and an inner simultaneous induction on the two arguments. Noting first that if $\mathsf{treduce}_A(x.\,R)$ is defined, it is a subterm of A, the fact that the recurrence relations respect this induction order can be verified almost by inspection. The only slightly subtle case is the equation for $\mathsf{reduce}_A(x.\,R\;N, N_0)$, which is the only case in which the subscripting type changes. Here the side condition $\mathsf{treduce}_A(x.\,R) \equiv \Pi x\!:\!B.\,C$ ensures that B must be a strict subterm of A for the reduction to be defined. An instantiation such as $\mathsf{inst\_n}_A(x.\,x\;x, \lambda x.\,x\;x)$ is guaranteed to fail the side condition after only finitely many expansions of the recurrence.

Another way in which an instance of the instantiation operators might fail to be defined would be if the recursive instantiation $\mathsf{inst\_r}_A(x.\,R, N_0)$ in the same equation failed to result in a manifest lambda abstraction $\lambda y.\,N'$. In fact, this could only happen if the term N_0 failed to have the ascribed type A.[5] So instantiation always terminates, regardless of whether its arguments are well typed, but it is not defined in all cases. After the meta-theory is further developed, it can be shown that instantiation is always defined on well-typed terms when the types match in the appropriate way.

No substantially new issues arise in the extension to the LLF fragment.

[5] Or a substitution instance of A.

DEFINITION 13 (INSTANTIATION, LLF FRAGMENT)

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

$$\begin{array}{ll} \mathsf{treduce}_A(x.\,R^\wedge N) \equiv C & \text{if } \mathsf{treduce}_A(x.\,R) \equiv B \multimap C \\ \mathsf{treduce}_A(x.\,\pi_1 R) \equiv B_1 & \text{if } \mathsf{treduce}_A(x.\,R) \equiv B_1 \mathbin{\&} B_2 \\ \mathsf{treduce}_A(x.\,\pi_2 R) \equiv B_2 & \text{if } \mathsf{treduce}_A(x.\,R) \equiv B_1 \mathbin{\&} B_2 \end{array}$$

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

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

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

$$\begin{array}{l} \mathsf{inst\_r}_A(x.\,R^\wedge N, N_0) \equiv (\mathsf{inst\_r}_A(x.\,R, N_0))^\wedge(\mathsf{inst\_r}_A(x.\,N, N_0)) \\ \mathsf{inst\_r}_A(x.\,\pi_1 R, N_0) \equiv \pi_1(\mathsf{inst\_r}_A(x.\,R, N_0)) \\ \mathsf{inst\_r}_A(x.\,\pi_2 R, N_0) \equiv \pi_2(\mathsf{inst\_r}_A(x.\,R, N_0)) \end{array}$$

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

$$\begin{array}{l} \mathsf{inst\_n}_A(x.\,\hat{\lambda} y.\,N, N_0) \equiv \hat{\lambda} y.\,\mathsf{inst\_n}_A(x.\,N, N_0) \quad \text{if } y \notin \mathrm{FV}(N_0) \\ \mathsf{inst\_n}_A(x.\,\langle N_1, N_2 \rangle, N_0) \equiv \langle \mathsf{inst\_n}_A(x.\,N_1, N_0), \mathsf{inst\_n}_A(x.\,N_2, N_0) \rangle \\ \mathsf{inst\_n}_A(x.\,\langle\rangle, N_0) \equiv \langle\rangle \end{array}$$

In order to extend this idea to the full CLF language, with its pattern-oriented destructor for the monadic type, it is necessary to introduce *matching operators* $\mathsf{match\_}c_S(p.\,E, X)$, where X is either an expression or a monadic object. The matching operator computes the result of instantiating E according to the substitution on the variables of p generated by matching p against X. (The variables in p should be considered bound in E.) In the case that X is a monadic object M_0, this is straightforward: the syntax of monadic objects corresponds precisely to that of patterns. But in the case that X is a let binding, an interesting issue arises:

$$\mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2) \equiv\ ?$$

The key is found in Pfenning and Davies' *non-standard substitutions* for the proof terms of the modal logics of possibility and laxity [PD01]. These analyze the structure of the object being substituted, not, as in the usual case, the term being substituted into. The effect is similar to a commuting conversion:

$$\begin{array}{l} \mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2) \equiv \\ \qquad (\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, E_2)) \end{array}$$

It is interesting that both non-standard substitution and pattern matching—the latter not present in Pfenning and Davies' system—rely in this way on an analysis of the object being substituted rather than the term being substituted into. In a sense, this commonality is what makes the harmonious interaction between CLF's modality and its synchronous types possible.

Definition 14 (Instantiation, full CLF)

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

$$\mathsf{inst\_n}_A(x.\,\{E\}, N_0) \equiv \{\mathsf{inst\_e}_A(x.\,E, N_0)\}$$

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

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

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

$$\begin{aligned}
&\mathsf{inst_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv (\mathsf{let}\ \{p\} = \mathsf{inst_r}_A(x.\,R, N_0)\ \mathsf{in}\ \mathsf{inst_e}_A(x.\,E, N_0))\\
&\qquad \text{if } \mathrm{head}(R) \text{ is not } x,\\
&\qquad \text{and } \mathrm{FV}(p) \cap \mathrm{FV}(N_0) \text{ is empty}\\
&\mathsf{inst_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv \mathsf{match_e}_S(p.\,\mathsf{inst_e}_A(x.\,E, N_0), E')\\
&\qquad \text{if } \mathsf{treduce}_A(x.\,R) \equiv \{S\},\ \mathsf{reduce}_A(x.\,R, N_0) \equiv \{E'\},\\
&\qquad \text{and } \mathrm{FV}(p) \cap \mathrm{FV}(N_0) \text{ is empty}\\
&\mathsf{inst_e}_A(x.\,M, N_0) \equiv \mathsf{inst_m}_A(x.\,M, N_0)
\end{aligned}$$

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

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

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

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

$\mathsf{inst\_s}_A(x.\,S, N_0) \equiv S'$ [Synchronous type instantiation]

(Analogous.)

We interpret these recurrences as inductive definitions (adopting the least solution to the recurrence). The first theorem ensures that type reduction has been properly defined. There are also two lemmas, one of which, mentioned earlier, ensures that type reduction makes a type smaller. All are immediate by structural induction on the argument.

THEOREM 1 (DEFINABILITY OF TYPE REDUCTION)
The recurrence for type reduction uniquely determines a least partial function solving the recurrence.

LEMMA 2
If $\mathsf{treduce}_A(x.\,R)$ *is defined, then* $\mathrm{head}(R)$ *is* x.

LEMMA 3
If $\mathsf{treduce}_A(x.\,R) \equiv B$*, then* B *is a subterm of* A.

Now we can conclude that instantiation has been properly defined.

THEOREM 4 (DEFINABILITY OF INSTANTIATION)
The recurrence for the reduction, instantiation, and matching operators uniquely determines the least partial functions (up to α*-equivalence) solving them.*

Proof. The proof is by an outer structural induction on the type subscript, and an inner simultaneous structural induction on the two arguments. □

4.2 Expansion

The expansion operator is specified by the following equations. In some cases, new bound variables are introduced on the right-hand side of an equation. Any new variables in an instance of such an equation are required to be distinct from one another and from any other variables in the equation instance.

DEFINITION 15 (EXPANSION)
$\mathsf{expand}_A(R) \equiv N$ [Expansion]

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

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

$$
\begin{array}{l}
\mathsf{pexpand}_{S_1 \otimes S_2}(p_1 \otimes p_2) \equiv \mathsf{pexpand}_{S_1}(p_1) \otimes \mathsf{pexpand}_{S_2}(p_2) \\
\mathsf{pexpand}_1(1) \equiv 1 \\
\mathsf{pexpand}_{\exists x:A.S}([x, p]) \equiv [\mathsf{expand}_A(x), \mathsf{pexpand}_S(p)] \\
\mathsf{pexpand}_A(x) \equiv \mathsf{expand}_A(x)
\end{array}
$$

THEOREM 5 (DEFINABILITY OF EXPANSION)

1. *If* $\mathsf{pexpand}_S(p_1)$ *and* $\mathsf{pexpand}_S(p_2)$ *are both defined then* p_1 *and* p_2 *are the same up to variable renaming.*
2. *Given* S*, there is a pattern* p*, fresh with respect to any given set of variables, such that* $\mathsf{pexpand}_S(p)$ *is defined.*

3. *The recurrence for expansion uniquely determines it as a total function up to α-equivalence.*

Proof. The first part is by induction on S. The second and third parts are by induction on the type subscript, using the first part to ensure that the result of $\mathsf{expand}_{\{S\}}(R)$ is unique up to α-equivalence. □

4.3 On decidability

The next few meta-theoretic observations are on the decidability of the fundamental operators and judgments of the theory. We begin with equality. Recall that equality is totally independent of typing and is syntax directed.

THEOREM 6 (DECIDABILITY OF EQUALITY)

1. *Given R_1 and R_2, it is decidable whether $R_1 = R_2$.*
2. *Given N_1 and N_2, it is decidable whether $N_1 = N_2$.*
3. *Given M_1 and M_2, it is decidable whether $M_1 = M_2$.*
4. *Given E_1 and E_2, it is decidable whether $E_1 =_{\mathrm{c}} E_2$.*
5. *Given E_1 and E_2, it is decidable whether $E_1 = E_2$.*
6. *Given P_1 and P_2, it is decidable whether $P_1 = P_2$.*

Proof. The proof is by simultaneous structural induction on the subjects of the judgments. The judgments have been placed in the proper order of precedence in view of the trivial coercions from R to N to M to E. For example, $R_1 = R_2$ is below $N_1 = N_2$ only if each of R_1 and R_2 is a subterm of either N_1 or N_2, but $N_1 = N_2$ is below $R_1 = R_2$ only if each of N_1 and N_2 is a *strict* subterm of either R_1 or R_2. In the case of $E_1 =_{\mathrm{c}} E_2$ it suffices to note that any E_2 can be decomposed as $\epsilon[E_2']$ in only finitely many ways. □

The next theorems are on the decidability of instantiation and expansion. Instantiation and expansion are both syntax directed and terminate on arbitrary terms and type subscripts. There is no requirement that the terms or type subscript be valid or that the type subscript have any particular relationship to the terms. The proofs are immediate by the same induction schemes as for the definability theorems.

THEOREM 7 (DECIDABILITY OF INSTANTIATION)
It is decidable whether any instance of the instantiation and matching operators is defined, and if so, it can be effectively computed.

THEOREM 8 (DECIDABILITY OF EXPANSION)

1. *Given S, a pattern p can be effectively computed such that $\mathsf{pexpand}_S(p)$ is defined and can be effectively computed. The pattern p can be chosen fresh with respect to any given set of variables.*

2. *Given A and R, the result of $\mathsf{expand}_A(R)$ can be effectively computed.*

The fact that instantiation and equality are decidable leads directly to the decidability of typing in the framework, since the typing rules are syntax directed and appeal only to instantiation and equality.

Lemma 9 (Unicity of inference)

1. *There is a partial function* $\mathsf{typeof}(\Gamma; \Delta \vdash R)$ *such that whenever* $\Gamma; \Delta \vdash R \Rightarrow A$ *holds,* $A \equiv \mathsf{typeof}(\Gamma; \Delta \vdash R)$*. Given* Γ*,* Δ*, and* R*, it is decidable whether* $\mathsf{typeof}(\Gamma; \Delta \vdash R)$ *is defined, and if so, it can be effectively computed.*

2. *There is a partial function* $\mathsf{kindof}(\Gamma \vdash P)$ *such that whenever* $\Gamma \vdash P \Rightarrow K$ *holds,* $K \equiv \mathsf{kindof}(\Gamma \vdash P)$*. Given* Γ *and* P*, it is decidable whether* $\mathsf{kindof}(\Gamma \vdash P)$ *is defined, and if so, it can be effectively computed.*

Proof. Immediate by the facts that typing is syntax directed, instantiation is a partial function, and instantiation is computable. □

Lemma 10
Suppose that for all Γ' *and* Δ' *it is decidable whether* $\Gamma'; \Delta' \vdash E \leftarrow S$*. Then it is decidable whether* $\Gamma; \Delta; \Psi \vdash E \leftarrow S$*.*

Proof. The induction is on the number of type constructors in Ψ. □

Theorem 11 (Decidability of typing)

1. *Given* Γ*,* Δ*,* R*, and* A*, it is decidable whether* $\Gamma; \Delta \vdash R \Rightarrow A$*.*
2. *Given* Γ*,* Δ*,* N*, and* A*, it is decidable whether* $\Gamma; \Delta \vdash N \Leftarrow A$*.*
3. *Given* Γ*,* Δ*,* M*, and* S*, it is decidable whether* $\Gamma; \Delta \vdash M \Leftarrow S$*.*
4. *Given* Γ*,* Δ*,* E*, and* S*, it is decidable whether* $\Gamma; \Delta \vdash E \leftarrow S$*.*
5. *Given* Γ*,* P*, and* K*, it is decidable whether* $\Gamma \vdash P \Rightarrow K$*.*
6. *Given* Γ *and* A*, it is decidable whether* $\Gamma \vdash A \Leftarrow \mathsf{type}$*.*
7. *Given* Γ *and* K*, it is decidable whether* $\Gamma \vdash K \Leftarrow \mathsf{kind}$*.*
8. *Given* Σ*, it is decidable whether* $\vdash \Sigma\ \mathsf{ok}$*.*
9. *Given* Γ*, it is decidable whether* $\vdash \Gamma\ \mathsf{ok}$*.*
10. *Given* Γ *and* Δ*, it is decidable whether* $\Gamma \vdash \Delta\ \mathsf{ok}$*.*
11. *Given* Γ *and* Ψ*, it is decidable whether* $\Gamma \vdash \Psi\ \mathsf{ok}$*.*

Proof. The proof is by structural induction on the subject of each judgment. In order to decide $\Gamma; \Delta \vdash R \Leftarrow P$, we test whether $\mathsf{typeof}(\Gamma; \Delta \vdash R)$ is atomic, $\mathsf{typeof}(\Gamma; \Delta \vdash R) \equiv P$, and $\Gamma; \Delta \vdash R \Rightarrow \mathsf{typeof}(\Gamma; \Delta \vdash R)$. In order to decide $\Gamma; \Delta \vdash R\ N \Rightarrow A$, we first test whether $\mathsf{typeof}(\Gamma; \Delta \vdash R) \equiv \Pi x{:}B.C$ for some B and C, whether $\Gamma; \Delta \vdash R \Rightarrow \Pi x{:}B.C$, whether $\Gamma; \cdot \vdash N \Leftarrow B$, and finally whether $\mathsf{inst\_a}_B(x.C, N) \equiv A$. In order to decide $\Gamma; \Delta \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \leftarrow S$ there is an appeal to the lemma. Similar comments apply in other cases. In order to decide multiplicatives, it suffices—efficiency concerns aside—to test every possible decomposition of the linear context Δ into Δ_1, Δ_2. □

It is characteristic of this approach to the type theory that such decidability theorems can be proved before any of the other meta-theory is developed, and that they do not depend in any way on the various terms involved being valid. The most important reason is that every judgment and algorithm of the theory is syntax directed.

4.4 Composition

A novel element of the meta-theory is the need for composition theorems for the instantiation and expansion operators. These correspond to the usual categorical axioms of left and right identity and associativity (slightly modified because we instantiate a single variable at a time rather than all free variables). In a language admitting non-canonical forms these are trivial, since the identity principle is witnessed trivially, and there is a composition law for syntactic substitution:

$$[M_1/x_1][M_2/x_2]M_3 \equiv [[M_1/x_1]M_2][M_1/x_1]M_3$$

The main theorem of this section is a corresponding law for instantiation. We begin with a number of lemmas.

Lemma 12 (Trivial instantiation)

1. *If* $x \notin \mathrm{FV}(R)$*, then* $\mathsf{inst\_r}_A(x.\, R, N_0) \equiv R$.
2. *If* $x \notin \mathrm{FV}(N)$*, then* $\mathsf{inst\_n}_A(x.\, N, N_0) \equiv N$.
3. *If* $x \notin \mathrm{FV}(M)$*, then* $\mathsf{inst\_m}_A(x.\, M, N_0) \equiv M$.
4. *If* $x \notin \mathrm{FV}(E)$*, then* $\mathsf{inst\_e}_A(x.\, E, N_0) \equiv E$.
5. *If* $x \notin \mathrm{FV}(P)$*, then* $\mathsf{inst\_p}_A(x.\, P, N_0) \equiv P$.
6. *If* $x \notin \mathrm{FV}(B)$*, then* $\mathsf{inst\_a}_A(x.\, B, N_0) \equiv B$.
7. *If* $x \notin \mathrm{FV}(S)$*, then* $\mathsf{inst\_s}_A(x.\, S, N_0) \equiv S$.
8. *If* $x \notin \mathrm{FV}(K)$*, then* $\mathsf{inst\_k}_A(x.\, K, N_0) \equiv K$.

Proof. A straightforward induction on the first argument. □

Lemma 13
If $\mathsf{inst\_r}_A(x.\, R, N_0)$ *is defined, then* $\mathrm{head}(\mathsf{inst\_r}_A(x.\, R, N_0))$ *is* $\mathrm{head}(R)$.

Lemma 14
If x *and* y *are distinct,* $\mathsf{treduce}_A(x.\, R) \equiv A'$*, and* $\mathsf{inst\_r}_B(y.\, R, N_0)$ *is defined, then*

$$\mathsf{treduce}_A(x.\, \mathsf{inst\_r}_B(y.\, R, N_0)) \equiv A'.$$

Lemma 15
The free variables in a term resulting from instantiating x *with* N_0 *are a subset of the free variables of the original term, excepting* x.

The free variables in a term resulting from matching p *with* E_0 *or* M_0 *are a subset of the free variables of the original term, excepting the variables of* p.

Note that the following theorem holds without assuming that the terms involved are well-typed.

Theorem 16 (Composition of instantiations)
Suppose that x *and* y *are distinct or* $x \notin \mathrm{FV}(p)$*, and* $y \notin \mathrm{FV}(N_0)$ *or* $\mathrm{FV}(p) \cap \mathrm{FV}(N_0)$ *is empty, as the case may be. For each of the following equations, if the inner instantiations are defined, then the outermost instantiations on each side are both defined and are* α*-equivalent.*

1. $\mathsf{reduce}_{A_0}(x.\,\mathsf{inst\_r}_A(y.\,R_1, N_2), N_0) \equiv \mathsf{inst\_n}_A(y.\,\mathsf{reduce}_{A_0}(x.\,R_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
2. $\mathsf{inst\_n}_{A_0}(x.\,\mathsf{reduce}_A(y.\,R_1, N_2), N_0) \equiv \mathsf{reduce}_A(y.\,\mathsf{inst\_r}_{A_0}(x.\,R_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
3. $\mathsf{inst\_r}_{A_0}(x.\,\mathsf{inst\_r}_A(y.\,R_1, N_2), N_0) \equiv \mathsf{inst\_r}_A(y.\,\mathsf{inst\_r}_{A_0}(x.\,R_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
4. $\mathsf{inst\_n}_{A_0}(x.\,\mathsf{inst\_n}_A(y.\,N_1, N_2), N_0) \equiv \mathsf{inst\_n}_A(y.\,\mathsf{inst\_n}_{A_0}(x.\,N_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
5. $\mathsf{inst\_m}_{A_0}(x.\,\mathsf{inst\_m}_A(y.\,M_1, N_2), N_0) \equiv \mathsf{inst\_m}_A(y.\,\mathsf{inst\_m}_{A_0}(x.\,M_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
6. $\mathsf{inst\_e}_{A_0}(x.\,\mathsf{inst\_e}_A(y.\,E_1, N_2), N_0) \equiv \mathsf{inst\_e}_A(y.\,\mathsf{inst\_e}_{A_0}(x.\,E_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
7. $\mathsf{inst\_e}_{A_0}(x.\,\mathsf{match\_m}_S(p.\,E_1, M_2), N_0) \equiv \mathsf{match\_m}_S(p.\,\mathsf{inst\_e}_{A_0}(x.\,E_1, N_0), \mathsf{inst\_m}_{A_0}(x.\,M_2, N_0))$
8. $\mathsf{inst\_e}_{A_0}(x.\,\mathsf{match\_e}_S(p.\,E_1, E_2), N_0) \equiv \mathsf{match\_e}_A(p.\,\mathsf{inst\_e}_{A_0}(x.\,E_1, N_0), \mathsf{inst\_e}_{A_0}(x.\,E_2, N_0))$
9. $\mathsf{match\_m}_{S_0}(p_0.\,\mathsf{match\_e}_S(p.\,E_1, E_2), M_0)$
   $\equiv \mathsf{match\_e}_S(p.\,\mathsf{match\_m}_{S_0}(p_0.\,E_1, M_0), \mathsf{match\_m}_{S_0}(p_0.\,E_2, M_0))$
10. $\mathsf{match\_e}_{S_0}(p_0.\,\mathsf{match\_e}_S(p.\,E_1, E_2), E_0) \equiv \mathsf{match\_e}_S(p.\,E_1, \mathsf{match\_e}_{S_0}(p_0.\,E_2, E_0))$
    (Supposing no variable bound by p_0 is free in E_1.)
11. $\mathsf{inst\_p}_{A_0}(x.\,\mathsf{inst\_p}_A(y.\,P_1, N_2), N_0) \equiv \mathsf{inst\_p}_A(y.\,\mathsf{inst\_p}_{A_0}(x.\,P_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
12. $\mathsf{inst\_a}_{A_0}(x.\,\mathsf{inst\_a}_A(y.\,A_1, N_2), N_0) \equiv \mathsf{inst\_a}_A(y.\,\mathsf{inst\_a}_{A_0}(x.\,A_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
13. $\mathsf{inst\_s}_{A_0}(x.\,\mathsf{inst\_s}_A(y.\,S_1, N_2), N_0) \equiv \mathsf{inst\_s}_A(y.\,\mathsf{inst\_s}_{A_0}(x.\,S_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$
14. $\mathsf{inst\_k}_{A_0}(x.\,\mathsf{inst\_k}_A(y.\,K_1, N_2), N_0) \equiv \mathsf{inst\_k}_A(y.\,\mathsf{inst\_k}_{A_0}(x.\,K_1, N_0), \mathsf{inst\_n}_{A_0}(x.\,N_2, N_0))$

Proof. The proof has many cases but is straightforward. □

4.5 On equality

The following theorems show that the framework's equality is an equivalence relation.

THEOREM 17 (REFLEXIVITY OF EQUALITY)

1. *Given R, we have $R = R$.*
2. *Given N, we have $N = N$.*
3. *Given M, we have $M = M$.*
4. *Given E, we have $E =_{\mathrm{c}} E$.*
5. *Given E, we have $E = E$.*
6. *Given P, we have $P = P$.*

Proof. The proof is by structural induction on the subject of the judgment. □

LEMMA 18
If $E_1 =_{\mathrm{c}} (\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2')$, then $E_1 \equiv \epsilon[\mathsf{let}\ \{p\} = R_1\ \mathsf{in}\ E_1']$ for some R_1, ϵ and E_1' such that $R_1 = R_2$ and $\epsilon[E_1'] =_{\mathrm{c}} E_2'$. Furthermore, no variable free in R_1 is bound in ϵ.

Proof. The proof is by structural induction on the derivation of the assumption. □

THEOREM 19 (SYMMETRY OF EQUALITY)

1. *If $R_1 = R_2$, then $R_2 = R_1$.*
2. *If $N_1 = N_2$, then $N_2 = N_1$.*
3. *If $M_1 = M_2$, then $M_2 = M_1$.*

4. *If* $E_1 =_{\mathrm{c}} E_2$*, then* $E_2 =_{\mathrm{c}} E_1$*.*
5. *If* $E_1 = E_2$*, then* $E_2 = E_1$*.*
6. *If* $P_1 = P_2$*, then* $P_2 = P_1$*.*

Proof. The proof is by structural induction on the second subject of the assumed equality. In the case $E_1 =_{\mathrm{c}} (\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2')$ we appeal to the lemma. Then $E_2' =_{\mathrm{c}} \epsilon[E_1']$ and $R_2 = R_1$ by the induction hypothesis. Then $(\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2') =_{\mathrm{c}} \epsilon[\mathsf{let}\ \{p\} = R_1\ \mathsf{in}\ E_1'] \equiv E_1$ by an inference rule. □

In order to prove the transitivity of concurrent equality, we first need to consider a few trivial properties of concurrent contexts (somewhat tedious to prove syntactically).

Lemma 20
If $\epsilon[E_1] \equiv \epsilon[E_2]$*, then* $E_1 \equiv E_2$*.*

Proof. The result follows by induction on ϵ. □

Lemma 21
If $\epsilon_1[E] \equiv \epsilon_2[E]$*, then* $\epsilon_1 \equiv \epsilon_2$*.*

Proof. The result follows by induction on ϵ_1. □

Lemma 22
Suppose that no variable free in R *is bound in* ϵ_2*. If* $\epsilon_1[E_1] \equiv \epsilon_2[\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E_2]$*, then either*

1. $\epsilon_1 \equiv \epsilon_2[\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \epsilon']$ *for some* ϵ'*, or*
2. $E_1 \equiv \epsilon'[\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E_2]$ *for some* ϵ'*.*

Proof. The result follows by induction on ϵ_1. □

Now we prove a general inversion principle for concurrent equality.

Lemma 23 (Inversion principle for concurrent equality)
Suppose that no variable free in R_1 *is bound by* ϵ_1*. If* $\epsilon_1[\mathsf{let}\ \{p\} = R_1\ \mathsf{in}\ E_1'] =_{\mathrm{c}} E_2$*, then* $E_2 \equiv \epsilon_2[\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2']$ *for some* ϵ_2*,* R_2*, and* E_2' *such that* $R_1 = R_2$ *and* $\epsilon_1[E_1'] =_{\mathrm{c}} \epsilon_2[E_2']$*. Furthermore, no variable free in* R_2 *is bound by* ϵ_2*.*

Proof. The result follows by induction on ϵ_1. □

Now we can prove the transitivity theorem itself.

Theorem 24 (Transitivity of equality)

1. *If* $R_1 = R_2$ *and* $R_2 = R_3$*, then* $R_1 = R_3$*.*
2. *If* $N_1 = N_2$ *and* $N_2 = N_3$*, then* $N_1 = N_3$*.*
3. *If* $M_1 = M_2$ *and* $M_2 = M_3$*, then* $M_1 = M_3$*.*
4. *If* $E_1 =_{\mathrm{c}} E_2$ *and* $E_2 =_{\mathrm{c}} E_3$*, then* $E_1 =_{\mathrm{c}} E_3$*.*
5. *If* $E_1 = E_2$ *and* $E_2 = E_3$*, then* $E_1 = E_3$*.*
6. *If* $P_1 = P_2$ *and* $P_2 = P_3$*, then* $P_1 = P_3$*.*

Proof. The proof is by structural induction on the derivation of the first assumption. For the part involving concurrent equality we appeal to the inversion principle. □

It is the restriction to canonical forms inherent in the syntax of CLF that makes it possible to define equality and prove such a result without any reference to the typing judgments.

Now we go on to show that all of the primitive operators and judgments of the theory factor through the equivalence relation on well-typed terms induced by the equality judgment. (Recall that we do not ascribe any particular meaning to the equality judgment unless the terms involved are well-typed.) This licenses us to think of them as being defined on the equivalence classes.

First, however, we introduce a stronger version of concurrent equality, needed to stage the proofs. Strong concurrent equality only allows rearranging let bindings at the top level structure of an expression, rather than deep within it.

Definition 16 (Strong concurrent equality)
$E_1 =_{\mathrm{s}} E_2$ [Strong concurrent equality]

$$\frac{}{M =_{\mathrm{s}} M} \qquad \frac{E_1 =_{\mathrm{s}} \epsilon[E_2]}{(\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E_1) =_{\mathrm{s}} \epsilon[\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E_2]}\,*$$

Again the rule marked (*) is subject to the side condition that no variable bound by p be free in the conclusion or bound by the context ϵ, and that no variable free in R be bound by the context ϵ. We have that strong concurrent equality is reflexive, symmetric, and transitive, by essentially the same arguments as for the original concurrent equality.

Theorem 25
Strong concurrent equality is an equivalence relation.

Strong concurrent equality, as the name suggests, is a special case of concurrent equality. This follows from the reflexivity of framework equality.

Theorem 26
If $E_1 =_{\mathrm{s}} E_2$, then $E_1 =_{\mathrm{c}} E_2$.

We also have the following lemmas.

Lemma 27
Given S, p, E, ϵ, and E_0, we have $\mathsf{match\_e}_S(p.\,E, \epsilon[E_0]) \equiv \epsilon[\mathsf{match\_e}_S(p.\,E, E_0)]$, supposing one or the other side is defined.

Proof. The result follows by induction on ϵ. □

Lemma 28
Suppose that p and p' bind disjoint variables, and no variable free in R' is bound by p, and no variable free in E_0 or M_0 is bound by p'. Then the following equations hold, assuming one or the other side is defined.

1. $\mathsf{match\_m}_S(p.\,\mathsf{let}\ \{p'\} = R'\ \mathsf{in}\ E, M_0) \equiv (\mathsf{let}\ \{p'\} = R'\ \mathsf{in}\ \mathsf{match\_m}_S(p.\,E, M_0))$
2. $\mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p'\} = R'\ \mathsf{in}\ E, E_0) =_{\mathrm{s}} (\mathsf{let}\ \{p'\} = R'\ \mathsf{in}\ \mathsf{match\_e}_S(p.\,E, E_0))$

Proof. The first part follows by induction on S, and then the second part follows by induction on E_0. □

LEMMA 29
Suppose that p_1 and p_2 bind disjoint variables, and that no variable free in E_1 is bound by p_2, and no variable free in E_2 is bound by p_1. Then

$$\mathsf{match\_e}_{S_1}(p_1.\,\mathsf{match\_e}_{S_2}(p_2.\,E, E_2), E_1) =_{\mathrm{s}} \mathsf{match\_e}_{S_2}(p_2.\,\mathsf{match\_e}_{S_1}(p_1.\,E, E_1), E_2)$$

as long as one or the other side is defined.

Proof. The proof is by induction on E_1. In the case $E_1 \equiv M_1$ we appeal to the composition law for instantiation. Otherwise we appeal to the preceding lemma. □

The utility of strong concurrent equality is that the following theorem can be proved immediately.

THEOREM 30
In each of the following cases, the resulting equality holds assuming that one side or the other is defined.

1. *If no variable free in R is bound by ϵ, and ϵ and p bind disjoint sets of variables, then* $\mathsf{inst\_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \epsilon[E], N_0) =_{\mathrm{s}} \mathsf{inst\_e}_A(x.\,\epsilon[\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E], N_0)$.
2. *If $E =_{\mathrm{s}} E'$, then* $\mathsf{inst\_e}_A(x.\,E, N_0) =_{\mathrm{s}} \mathsf{inst\_e}_A(x.\,E', N_0)$.
3. *If $E =_{\mathrm{s}} E'$, then* $\mathsf{match\_m}_S(p.\,E, M_0) =_{\mathrm{s}} \mathsf{match\_m}_S(p.\,E', M_0)$.
4. *If $E =_{\mathrm{s}} E'$, then* $\mathsf{match\_e}_S(p.\,E, E_0) =_{\mathrm{s}} \mathsf{match\_e}_S(p.\,E', E_0)$.

Proof. The proof is by an outer induction on the type subscript. The first part also uses an inner induction on ϵ, the second part uses an inner induction on the derivation of $E =_{\mathrm{s}} E'$, and the last part uses an inner induction on E_0. □

We are now in a position to prove the "functionality" of instantiation with respect to equivalence classes modulo the framework's equality. We extend equality to types and kinds by the natural congruence rules.

THEOREM 31 (FUNCTIONALITY FOR INSTANTIATION)
In each of the following cases, the resulting equality holds assuming that one side or the other is defined.

1. *If $R = R'$ and $N_0 = N_0'$, then* $\mathsf{reduce}_A(x.\,R, N_0) = \mathsf{reduce}_A(x.\,R', N_0')$.
2. *If $R = R'$ and $N_0 = N_0'$, then* $\mathsf{inst\_r}_A(x.\,R, N_0) = \mathsf{inst\_r}_A(x.\,R', N_0')$.
3. *If $N = N'$ and $N_0 = N_0'$, then* $\mathsf{inst\_n}_A(x.\,N, N_0) = \mathsf{inst\_n}_A(x.\,N', N_0')$.
4. *If $M = M'$ and $N_0 = N_0'$, then* $\mathsf{inst\_m}_A(x.\,M, N_0) = \mathsf{inst\_m}_A(x.\,M', N_0')$.
5. *If $E = E'$ and $N_0 = N_0'$, then* $\mathsf{inst\_e}_A(x.\,E, N_0) = \mathsf{inst\_e}_A(x.\,E', N_0')$.
6. *If $E = E'$ and $M_0 = M_0'$, then* $\mathsf{match\_m}_S(p.\,E, M_0) = \mathsf{match\_m}_S(p.\,E', M_0')$.
7. *If $E = E'$ and $E_0 = E_0'$, then* $\mathsf{match\_e}_S(p.\,E, E_0) = \mathsf{match\_e}_S(p.\,E', E_0')$.
8. *If $P = P'$ and $N_0 = N_0'$, then* $\mathsf{inst\_p}_A(x.\,P, N_0) = \mathsf{inst\_p}_A(x.\,P', N_0')$.

9. *If* $B = B'$ *and* $N_0 = N_0'$, *then* $\mathsf{inst\_a}_A(x.\,B, N_0) = \mathsf{inst\_a}_A(x.\,B', N_0')$.
10. *If* $S = S'$ *and* $N_0 = N_0'$, *then* $\mathsf{inst\_s}_A(x.\,S, N_0) = \mathsf{inst\_s}_A(x.\,S', N_0')$.
11. *If* $K = K'$ *and* $N_0 = N_0'$, *then* $\mathsf{inst\_k}_A(x.\,K, N_0) = \mathsf{inst\_k}_A(x.\,K', N_0')$.

Proof. The proof is by an outer induction on the type subscript and an inner simultaneous induction on the derivations of the two assumed equalities.

In most cases the result follows immediately by congruence rules. The critical parts are the ones given by induction over expressions. For $\mathsf{match\_e}_S(p.\,E, E_0) =_{\mathrm{c}} \mathsf{match\_e}_S(p.\,E', E_0')$ we appeal to Lemma 27.

For $\mathsf{inst\_e}_A(x.\,E, N_0) =_{\mathrm{c}} \mathsf{inst\_e}_A(x.\,E', N_0')$ we reason as follows. If $E \equiv M$, the result is immediate by the induction hypothesis. Otherwise $E \equiv (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E_1)$, $E' \equiv \epsilon[\mathsf{let}\ \{p\} = R'\ \mathsf{in}\ E_1']$, $R = R'$, and $E_1 =_{\mathrm{c}} \epsilon[E_1']$. Then

$$\mathsf{inst\_e}_A(x.\,E_1, N_0) =_{\mathrm{c}} \mathsf{inst\_e}_A(x.\,\epsilon[E_1'], N_0')$$

by the induction hypothesis. It follows that

$$\mathsf{inst\_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E_1, N_0) =_{\mathrm{c}} \mathsf{inst\_e}_A(x.\,\mathsf{let}\ \{p\} = R'\ \mathsf{in}\ \epsilon[E_1'], N_0')$$

by another appeal to the induction hypothesis. But by the preceding theorem,

$$\mathsf{inst\_e}_A(x.\,\mathsf{let}\ \{p\} = R'\ \mathsf{in}\ \epsilon[E_1'], N_0') =_{\mathrm{s}} \mathsf{inst\_e}_A(x.\,\epsilon[\mathsf{let}\ \{p\} = R'\ \mathsf{in}\ E_1'], N_0').$$

The result then follows by transitivity. □

A similar result holds for the expansion operator.

Theorem 32 (Functionality for expansion)
If $R = R'$, *then* $\mathsf{expand}_A(R) = \mathsf{expand}_A(R')$.

Proof. The proof is by induction on A. □

The functionality of instantiation immediately leads to a similar result for the typeof and kindof operators of Lemma 9. We extend equality to contexts and signatures in the obvious way.

Lemma 33
If $\Sigma = \Sigma'$, $\Gamma = \Gamma'$, *and* $\Delta = \Delta'$, *then* $\mathsf{typeof}(\Gamma;\ \Delta \vdash_\Sigma R) = \mathsf{typeof}(\Gamma';\ \Delta' \vdash_{\Sigma'} R)$ *and* $\mathsf{kindof}(\Gamma \vdash_\Sigma P) = \mathsf{kindof}(\Gamma' \vdash_{\Sigma'} P)$, *supposing one or the other side is defined.*

It is characteristic of the syntax-directed approach that the following theorem holds even when the contexts, etc. are not valid.

Theorem 34 (Type conversion)
Suppose that $\Sigma = \Sigma'$, $\Gamma = \Gamma'$, $\Delta = \Delta'$, $A = A'$, *and* $S = S'$.

1. $\Gamma;\ \Delta \vdash_\Sigma R \Rightarrow \mathsf{typeof}(\Gamma;\ \Delta \vdash_\Sigma R)$ *iff* $\Gamma';\ \Delta' \vdash_{\Sigma'} R \Rightarrow \mathsf{typeof}(\Gamma';\ \Delta' \vdash_{\Sigma'} R)$.
2. $\Gamma;\ \Delta \vdash_\Sigma N \Leftarrow A$ *iff* $\Gamma';\ \Delta' \vdash_{\Sigma'} N \Leftarrow A'$.
3. $\Gamma;\ \Delta \vdash_\Sigma M \Leftarrow S$ *iff* $\Gamma';\ \Delta' \vdash_{\Sigma'} M \Leftarrow S'$.
4. $\Gamma;\ \Delta \vdash_\Sigma E \leftarrow S$ *iff* $\Gamma';\ \Delta' \vdash_{\Sigma'} E \leftarrow S'$.

5. $\Gamma \vdash_\Sigma P \Rightarrow \mathsf{kindof}(\Gamma \vdash_\Sigma P)$ *iff* $\Gamma' \vdash_{\Sigma'} P \Rightarrow \mathsf{kindof}(\Gamma' \vdash_{\Sigma'} P)$.
6. $\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type}$ *iff* $\Gamma' \vdash_{\Sigma'} A \Leftarrow \mathsf{type}$.
7. $\Gamma \vdash_\Sigma S \Leftarrow \mathsf{type}$ *iff* $\Gamma' \vdash_{\Sigma'} S \Leftarrow \mathsf{type}$.
8. $\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind}$ *iff* $\Gamma' \vdash_{\Sigma'} K \Leftarrow \mathsf{kind}$.

Proof. The theorem is proved by induction over the assumed typing derivation, using Lemma 9. □

4.6 On typing

Now we can go on to prove the substitution and identity principles for CLF, and that they are witnessed by the instantiation and expansion operators. The instantiation operators are extended to contexts in the obvious way. Again, the theorem holds even when the contexts are not valid.

Theorem 35 (Substitution principles)
If $\Gamma_\mathrm{L}; \cdot \vdash N_0 \Leftarrow A$ *is derivable,* $\Gamma_\mathrm{L}, x{:}A, \Gamma_\mathrm{R}; \Delta \vdash N \Leftarrow C$ *is derivable,* $\mathsf{inst\_a}_A(x.\,\Gamma_\mathrm{R}, N_0) \equiv \Gamma'_\mathrm{R}$, $\mathsf{inst\_a}_A(x.\,\Delta, N_0) \equiv \Delta'$, *and* $\mathsf{inst\_a}_A(x.\,C, N_0) \equiv C'$, *the following hold:*

1. *The instantiation* $\mathsf{inst\_n}_A(x.\,N, N_0)$ *is defined.*
2. *The judgment* $\Gamma_\mathrm{L}, \Gamma'_\mathrm{R}; \Delta' \vdash \mathsf{inst\_n}_A(x.\,N, N_0) \Leftarrow C'$ *is derivable.*

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

1. *The instantiation* $\mathsf{inst\_n}_A(x.\,N, N_0)$ *is defined.*
2. *The judgment* $\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{inst\_n}_A(x.\,N, N_0) \Leftarrow C$ *is derivable.*

This result is mutually dependent on many other substitution theorems for the other syntactic categories. Space considerations preclude the incorporation of the proof here. It is notable that although the theorem as a whole does not assume the contexts are valid, in each case "just enough" information about the validity of various terms is available in order to make the induction go through.

Note that the presence of the type subscript to the instantiation operators is redundant if the objects involved are known to be well typed, since the types generated as reduce 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 an instantiation 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.

The following theorem can be proved by a simple structural induction.

Lemma 36 (Expansion)
If $\Gamma; \Delta \vdash R \Rightarrow A$ *then* $\Gamma; \Delta \vdash \mathsf{expand}_A(R) \Leftarrow A$.

As an immediate corollary, we have the following identity property.

Theorem 37 (Identity principles)
Any instance of $\Gamma, x{:}A; \cdot \vdash \mathsf{expand}_A(x) \Leftarrow A$ *or* $\Gamma; x \hat{:} A \vdash \mathsf{expand}_A(x) \Leftarrow A$ *is derivable.*

It is worth recalling that the substitution and identity principles are are needed to ensure that the type theory makes sense, since the syntatic restrictions inherent in CLF make it impossible to generate proofs of $A \to A$ or to compose proofs of $A \to B$ and $B \to C$ in any other way.

4.7 Related work

With one exception [Fel91], prior presentations of LF and LLF have been based on a syntax in which not every term is canonical. A difficulty is that equality cannot then be axiomatized in a manifestly decidable, syntax-directed way. In their original presentation of LF, Harper et al. define equality in terms of β-convertibility, and do not address η-conversions [HHP93]. Strong normalization ensures that this notion of equality is decidable. However, the η-conversions pose a special difficulty because of the lack of confluence for $\beta\eta$-reduction in the case of non-well-typed terms. Since LF typing is dependent on equality, the attempt to define an equality based on $\beta\eta$-reduction leads to a Catch-22.

Coquand [Coq91] tests $\beta\eta$-convertibility in LF using untyped β-reduction and extensionality, which is applied when comparing a λ-abstraction to a non-abstraction. However, this method fails when a unit type is present—as in LLF—because it may be necessary to apply extensionality even when neither of the terms being compared is a manifest unit introduction.

Cervesato's presentation of LLF avoids the η-conversion problem by restricting the syntax to η-long terms [Cer96]. The equality of the framework is still defined in terms of β-reduction. This is possible because the β-reducts of η-long terms are η-long.

Goguen proposes an elegant theory based on a typed notion of reduction [Gog94, Gog99]. An operational semantics based on this typed reduction is then shown to be decidable, and equivalent to a (not manifestly decidable) axiomatization of equality from first principles such as extensionality. However, being based on η-reduction, this approach does not decide equality by an analysis of canonical forms, which are η-long. This conflicts with the LF representation methodology, which emphasizes the primacy of canonical forms in constructing representations and proving their adequacy.

Ghani [Gha97] uses a typed rewriting relation similar to Goguen's operational semantics but with η-expansion rather than η-reduction. This leads to a more pleasant theory, especially given that normal forms with respect to Ghani's rewrite rule are canonical. Harper and Pfenning [HP00] also adopt an approach similar to Goguen's, in that equality is defined axiomatically and shown to be equivalent to a decision procedure. Their method improves on Goguen's in that the decision procedure is based on transforming a pair of terms simultaneously into canonical form. It offers the further advantage that the transformation into canonical form is incremental and can be aborted as soon as it is evident that the canonical forms of the two terms being compared will not be the same, an important concern for efficient implementation.

Felty has described a *canonical LF* in which only canonical forms are well-typed [Fel91]. This offers a number of advantages over other approaches: equality itself need not be axiomatized at all, because terms are equal just when they are identical (up to α-equivalence). And the representation methodology has an attractive simplicity: one establishes a compositional bijection between object-language terms and LF objects of a given type. One need not restrict the range of the bijection to "canonical LF terms" because every term in canonical LF is "canonical." However, Felty's development falls back on untyped β-reduction in order to define the typing judgment on canonical terms, so a syntax of non-canonical terms ends up being reintroduced after all, and strong normalization and confluence for the non-canonical forms must be proved. Thus, the canonical language cannot be considered foundational.

Relative to these prior developments, a contribution of this work is to elaborate a foundation for LF and LLF that preserves the attractive features of Felty's canonical LF, while eliminating entirely its dependence on non-canonical terms and β-reduction, in favor of a instantiation operator taking canonical terms into canonical terms. The key observation is that the instantiation operator can be defined in a manifestly terminating, syntax-directed way, even over ill-typed terms. This essentially eliminates the mutual dependence of typing and equality—since extensionality principles depend on typing—that is inherent in Goguen's or Harper and Pfenning's work. However, it is important to stress that this approach provides only a *foundation*. An efficient implementation would need to reintroduce defined constants and explicit substitutions, each of which would make the equality on the LF fragment non-trivial again. But in contrast to previous approaches, the framework is defined without any reference to such "non-canonical" forms: it can scale up to include them, but its foundation is independent of them.

5 Overview of proposed work

I propose to develop the theory and implementation of the concurrent logical framework CLF to the point that the advantages of having intrinsic support for representing concurrent systems in one's logical framework have been demonstrated. The development of a representation methodology for CLF and the implementation of the system should follow closely the example set by the Elf language [Pfe94a] and its implementation in the Twelf system [Pfe98]. The tasks include extending the current understanding of the theory of CLF, notably in considering the matching and unification problems; constructing a set of examples that flesh out the representation methodology for concurrent systems; understanding how the framework can be endowed with a logic programming interpretation that extends the Elf language and enables interesting computations based on the representations; and investigating whether the support for concurrency enables simpler meta-theoretic reasoning. On the implementation side, the tasks will include implementing a CLF checker to demonstrate its feasability and exploring potential strategies for a full-fledged mechanization of the operational semantics.

My work to date has established, together with my collaborators, a promising design for the framework itself, and the meta-theory of the candidate design has been worked out in detail [WCPW02]. It is possible that steps of the program outlined above not yet completed may reveal weaknesses in the current reference design for CLF. However, the risk that any needed changes would put the project seriously off track is mitigated by the inherently modular nature of the design and the key technical decision to base the meta-theory entirely on canonical forms. Unlike prior approaches based on analyses of $\beta\eta$-conversion, the instantiation operator technique is quite flexible and insensitive to changes in the details of the framework.

My collaborators have also developed a number of specifications using the current CLF design, including the operational semantics of an ML variant with recursive types, parametric polymorphism, suspensions, references, concurrency in the style of CML [Rep99] and futures in the style of Multilisp [Hal85]; Petri nets with labeled tokens; and synchronous and asynchronous variants of the π-calculus [Mil99]. Adequacy results have also been proved for the latter two specifications [CPWW02]. These examples will need to be re-evaluated in light of the logic programming interpretation that is eventually developed for CLF: it is not uncommon for adequacy results to be indifferent to small changes that nevertheless have an enormous impact on execution performance. Additional examples that could increase our confidence in the representation methodology include action calculi [Mil96b, BGHP97]; operational semantics for other concurrent programming languages; the Forum language [Mil96a, Chi95]; security protocol specification languages such as MSR [Cer01]; and proof nets [Gir87].

5.1 Framework extensions

I believe several interesting extensions to the framework could be integrated smoothly into the existing theory. First, it would be pleasant to extend the synchronous type language to include additive sum types $S_1 \oplus S_2$ and the additive unit type 0. The expanded type theory would include every connective of intuitionistic linear logic. This would make a satisfying end to the process by which more and more connectives of ILL have been incorporated into the framework in passing from LF to LLF to CLF.

Second, it is difficult to imagine an efficient implementation that did not allow the user to establish *notational definitions* [PS98] in a signature. In effect, notational definitions would allow the user to recover non-canonical terms in a controlled way. In fact, in a survey of the many example LF specifications included in the Twelf distribution, one finds that the *only* non-canonical aspect of these examples—aside from trivial η-conversions—lies in the use of notational definitions. LF specifications, as a rule, do not include non-β-normal terms.

Finally, it is well known that the efficient implementation of logical frameworks depends on the ability to perform operations on terms incrementally. The Twelf implementation extends the LF framework with a calculus of *explicit substitutions* [DHKP96], which allow equality testing, matching, and unification to be performed in an incremental way. Explicit substitutions also lead to a particularly clean description and implementation of the higher-order unification problem. In the CLF setting, adding a notion of explicit substitutions would require modifications similar to those needed to incorporate notational definitions. Equality testing, for example, would involve incrementally expanding notational definitions and explicit substitutions (via a suitably generalized reduction operator), in effect lazily recovering the underlying canonical terms being compared.

5.2 On the proof search problem

Much of the usefulness of the LF family of logical frameworks stems from the fact that LF specifications admit a logic programming interpretation that generalizes λ-Prolog, roughly speaking [NM88]. The Elf language is a concrete embodiment of the LF type theory with an operational semantics based on controlled search for terms of a given type (or type schema). Mapping through the fundamental representation principles for LF, *judgments as types* and *deductions as objects*, the operational semantics mechanizes proof search and related tasks in all of the enormous range of systems that are amenable to such representations. This interpretation has been extended to the LLF fragment, though the picture grows somewhat more complex [HM94, Cer96, CHP96, HP97].

In order to endow CLF with an operational semantics conservatively extending Elf's, it will be essential to clarify the role that the concurrent equality plays in collapsing the search space. Will it be possible to give the operational semantics its own concurrency, mirroring the concurrency at the level of the specification? How can the non-determinism associated with independent steps of a concurrent computation be resolved independently? The answers to these questions are as yet unknown. It may be that techniques developed for concurrent constraint languages related to Prolog can be adapted to this purpose. Prior work on the operational semantics of Forum and ACL may be relevant, as well as Harland's work on forward chaining in linear logic proof search, which raises quite similar issues [HPW00].

5.3 On the matching and unification problems

The ability to decide whether any two objects are equal is a prerequisite for the framework to be implemented at all. But many tasks which commonly arise in implementations of a framework, such as the reconstruction of omitted type information, or proof search, require generalizations of the equality decision procedure in the case that the objects being compared contain free (existential) variables. The result of the generalized procedure is either "yes, there is a most general instantiation of the free variables making the objects equal," or "no, there is no instantiation making them equal," or perhaps "maybe—please wait a while and try again."

The most familiar example of such a generalized decision procedure is the (higher-order) unification algorithm [Hue75]. In the Twelf implementation, the third "maybe" answer arises in the case that the higher-order unification algorithm faces a disjunctive non-deterministic choice. According to the task being performed, this may cause the overall task to fail, or else the unification problem might be postponed as a constraint, in the hope that further information might be able to refine the answer. Since higher-order unification is undecidable, any terminating algorithm must answer "maybe" in some cases.

But there are simpler instances of the generalized equality decision problem which, on the LF language, do not suffer from disjunctive non-determinism. An example is the *matching problem*, in which a ground object (containing no free variables) is compared to a pattern object, which contains free variables restricted to the *higher-order pattern fragment* [Mil91, PP03]. Many applications of the logic programming interpretation of proof search only make use of matching rather than the full generality of unification. So it would be nice to be able to describe a "pattern fragment" and an matching algorithm for CLF, even if the unification problem turned out to be intractable. However, it is not clear even for LLF what the "pattern fragment" should be, though there is some prior work on unification for LLF [CP97].

Unfortunately, the fact that CLF's equality admits permutative conversions adds a new source of non-determinism to the matching problem:

$$\begin{aligned}(\mathsf{let}\ \{1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{1\} = R_2\ \mathsf{in}\ 1) &\doteq (\mathsf{let}\ \{1\} = X_1\ \mathsf{in}\ \mathsf{let}\ \{1\} = X_2\ \mathsf{in}\ 1)\\ &\longrightarrow (X_1 \doteq R_1 \wedge X_2 \doteq R_2) \vee (X_1 \doteq R_2 \wedge X_2 \doteq R_1)\end{aligned}$$

A tantalizing question is whether such non-determinism is truly disjunctive. Or is there an interpretation of the matching problem on CLF expressions that is only guaranteed to succeed if some *arbitrary* permutation of the one expression can be matched with the other? This would reinterpret the permutative non-determinism as "don't-care" (conjunctive) rather than "don't-know" (disjunctive). I hope to be able to answer this question by considering what interpretation of the matching problem seems to make sense for proof search over the example CLF specifications that have been developed.

5.4 On representing meta-proofs

An additional area of application for LF-style frameworks is reasoning at a higher level: proving meta-theorems about a representation. For LF specifications there is a methodology for representing the computational content of meta-theorems by encoding the relation that the theorem embodies as a type [PR92, Pfe93, Pfe00a, Pfe00b]. The computational content of the theorem is then accessible via the logic programming interpretation of LF. The well-typedness of the relational representation does not, however, guarantee that the theorem actually holds. Additional conditions that must be verified include the *mode*, *termination*, and *coverage* properties of the relational encoding. These, in essence, ensure that the encoded relation actually determines a total function via the operational semantics. The Twelf system has incorporated mechanical checkers for each of these properties of LF relational encodings [RP96, Pie00, SP03], as well as an automated meta-theorem prover for a meta-logic formalizing the correctness of such encodings [SP98, Sch00].

For CLF the challenge will be to understand how relational encodings interact with the flexible definitional equality provided by the framework. Also, while LLF has been used to encode meta-theorems such as cut elimination for the linear sequent calculus [Pfe94b], the additional checks that would ensure that such LLF relational encodings constitute proofs have not been formalized. A possible contribution of the proposed work would be to clarify these issues.

6 Research plan

The most immediate tasks are to extend the framework with additive sum types and to augment its equational theory with another layer accounting for notational definitions and explicit substitutions. These require only minor modifications to the framework's meta-theory and do not depend on any other tasks. These extensions having been completed, it will be possible to implement a simple checker for the final, extended CLF language, a crucial demonstration of the framework's feasibility in practice. I am hopeful that much of the infrastructure of such a system could be borrowed from the existing Twelf system.

Concurrently, it will be necessary to formulate additional example CLF specifications. Once the checker has been completed, these specifications can be put on a more concrete basis and formally verified as to type correctness. They can also be examined in light of the question of what kind of operational semantics for proof search makes the most sense. As the range of examples grows and confidence builds in the suitability of the operational model chosen, it will be possible to formulate an initial operational semantics based on a static restriction avoiding the need for unification outside the LF fragment. (All the examples considered thus far satisfy such a restriction.) I have already elaborated a concrete description, not discussed here, of an operational semantics for the LLF fragment; this provides a starting point for the extension to full CLF.

The question of what notions of matching and unification are most suitable in the full CLF language cannot be addressed until examples depending on this full generality are formulated. These could include formal descriptions of interesting relations between concurrent computations, or meta-proofs regarding the operational semantics of a concurrent object language. I believe that matching offers useful and interesting capabilities even should the unification problem prove intractable.

Even should matching and unification both prove too unwieldy to implement effectively on full CLF, it should still be possible, once an operational semantics of proof search has been formulated, to implement a prototype proof search engine based on the restriction of unification to the LF fragment. Such an implementation would allow concrete experiments to be performed to help validate the design.

7 Conclusion

I am proposing to show that representations of concurrent systems can be succinctly and straightforwardly constructed using a logical framework with a notion of equality that models concurrency. The CLF framework as described above is a possible design for such a framework. I have shown that the framework is decidable and investigated its meta-theory. Much remains to do, but I am hopeful that the final result of this work will be a concrete language in which it is as unnecessary to specify or think about the low-level mechanics of the representation of concurrent computations as it would be to specify or think about matters of α-conversion and capture-avoiding substitution in LF. Furthermore, we would have a type-theoretic account in the style of LF of the full complement of connectives of intuitionistic linear logic, finally bringing the process of expansion that began with the extension of LF to LLF to its natural conclusion. And it is to be hoped that the direct, modular and scalable account of the type theory proposed here will provide a solid foundation for future explorations within the LF family of frameworks, of ideas such as proof irrelevance and ordered hypothetical judgments.

Acknowledgements. The CLF framework was developed in collaboration with Iliano Cervesato, Frank Pfenning and David Walker.

A The synchronous π-calculus

The following signature is a specification of synchronous π-calculus computations. It makes use of the Elf convention that free variables in a declaration are implicitly quantified by Π on the outside. Arguments corresponding to the implicit quantifiers are omitted. The representation function for processes is also shown.

Representation.

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

$$
\begin{array}{lcl}
\mathsf{par} & : & \mathsf{expr} \to \mathsf{expr} \to \mathsf{expr}. \\
\mathsf{new} & : & (\mathsf{chan} \to \mathsf{expr}) \to \mathsf{expr}. \\
\mathsf{rep} & : & \mathsf{expr} \to \mathsf{expr}. \\
\mathsf{sync} & : & \mathsf{sum} \to \mathsf{expr}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner P \mid Q \urcorner & = & \mathsf{par}\, \ulcorner P \urcorner\, \ulcorner Q \urcorner \\
\ulcorner \mathsf{new}\, u\, P \urcorner & = & \mathsf{new}\, (\lambda u. \ulcorner P \urcorner) \\
\ulcorner !P \urcorner & = & \mathsf{rep}\, \ulcorner P \urcorner \\
\ulcorner M \urcorner & = & \mathsf{sync}\, \ulcorner\!\ulcorner M \urcorner\!\urcorner
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{null} & : & \mathsf{sum}. \\
\mathsf{alt} & : & \mathsf{act} \to \mathsf{sum} \to \mathsf{sum}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner\!\ulcorner 0 \urcorner\!\urcorner & = & \mathsf{null} \\
\ulcorner\!\ulcorner c + M \urcorner\!\urcorner & = & \mathsf{alt}\, \ulcorner\!\ulcorner\!\ulcorner c \urcorner\!\urcorner\!\urcorner\, \ulcorner\!\ulcorner M \urcorner\!\urcorner
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{silent} & : & \mathsf{expr} \to \mathsf{act}. \\
\mathsf{in} & : & \mathsf{chan} \to (\mathsf{chan} \to \mathsf{expr}) \to \mathsf{act}. \\
\mathsf{out} & : & \mathsf{chan} \to \mathsf{chan} \to \mathsf{expr} \to \mathsf{act}.
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner\!\ulcorner\!\ulcorner \tau.P \urcorner\!\urcorner\!\urcorner & = & \mathsf{silent}\, \ulcorner P \urcorner \\
\ulcorner\!\ulcorner\!\ulcorner u(v).P \urcorner\!\urcorner\!\urcorner & = & \mathsf{in}\, u\, (\lambda v. \ulcorner P \urcorner) \\
\ulcorner\!\ulcorner\!\ulcorner \overline{u}\langle v \rangle.P \urcorner\!\urcorner\!\urcorner & = & \mathsf{out}\, u\, v\, \ulcorner P \urcorner
\end{array}
$$

Computations.

$$
\begin{array}{lcl}
\mathsf{proc} & : & \mathsf{expr} \to \mathsf{type}. \\
\mathsf{choice} & : & \mathsf{sum} \to \mathsf{type}. \\
\mathsf{select} & : & \mathsf{sum} \to \mathsf{act} \to \mathsf{type}.
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathsf{exit} & : & \mathsf{choice\ null} \multimap \{1\}. \\
\mathsf{internal} & : & \mathsf{choice}\, M \multimap \mathsf{select}\, M\, (\mathsf{silent}\, P) \to \{\mathsf{proc}\, P\}. \\
\mathsf{external} & : & \mathsf{choice}\, M_1 \multimap \mathsf{choice}\, M_2 \multimap \\
 & & \quad \mathsf{select}\, M_1\, (\mathsf{in}\, U\, (\lambda w\!:\!\mathsf{chan}.\, P\, w)) \to \\
 & & \quad \mathsf{select}\, M_2\, (\mathsf{out}\, U\, V\, Q) \to \\
 & & \quad \{\mathsf{proc}\, (\mathsf{par}\, (P\, V)\, Q)\}.
\end{array}
$$

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

References

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.

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

[AP91] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

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

[Bar97] Andrew Graham Barber. *Linear Type Theories, Semantics and Action Calculi.* PhD thesis, University of Edinburgh, 1997. Available as Technical Report ECS-LFCS-97-371.

[BGHP97] Andrew Barber, Philippa Gardner, Masahito Hasegawa, and Gordon D. Plotkin. From action calculi to linear logic. In M. Nielsen and W. Thomas, editors, *Proceedings of the 11th International Workshop on Computer Science Logic (CSL'97)*, pages 78–97, Aarhus, Denmark, August 1997. Springer-Verlag LNCS 1414.

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

[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.

[CHP96] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, March 1996. Springer-Verlag LNAI 1050.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[CP97] Iliano Cervesato and Frank Pfenning. Linear higher-order pre-unification. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science (LICS'97)*, pages 422–433, Warsaw, Poland, June 1997. IEEE Computer Society Press.

[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.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[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.

[FM97] M. Fairtlough and M.V. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, August 1997.

[Gha97] Neil Ghani. Eta-expansions in dependent type theory — the calculus of constructions. In P. de Groote and J.R. Hindley, editors, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 164–180, Nancy, France, April 1997. Springer-Verlag LNCS 1210.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gog94] Healfdene Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, Edinburgh University, August 1994.

[Gog99] Healfdene Goguen. Soundness of the logical framework for its typed operational semantics. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 177–197, L'Aquila, Italy, April 1999. Springer-Verlag LNCS 1581.

[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.

[HP97] James Harland and David Pym. Resource-distribution via boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer-Verlag LNAI 1249.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[HPW96] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, pages 391–405, Munich, July 1996. Springer-Verlag LNCS 1101.

[HPW00] James Harland, David Pym, and Michael Winikoff. Forward and backward chaining in linear logic. In D. Galmiche, editor, *Proceedings of the Workshop on Type-Theoretic Languages: Proof Search and Semantics*, volume 37 of *Electronic Notes in Theoretical Computer Science*, Pittsburgh, Pennsylvania, June 2000. Elsevier Science.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[HWTK98] Joshua S. Hodas, Kevin M. Watkins, Naoyuki Tamura, and Kyoung-Sun Kang. Efficient implementation of a linear logic programming language. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 145–159, Manchester, June 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.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[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.

[Mil96a] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[Mil96b] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[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.

[NM88] Gopalan Nadathur and Dale Miller. *Higher-Order Logic Programming*, volume 5 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter 8, pages 499–590. Oxford University Press, 1988.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, August 2001.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Per98] Guy Perrier. Concurrent programming as proof net construction. *Mathematical Structures in Computer Science*, 8(6):681–710, 1998.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[Pfe93] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 1993. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe94a] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe94b] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Pfe98] Frank Pfenning. Reasoning about deductions in linear logic. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 1–2, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421. Abstract for invited talk.

[Pfe99] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.

[Pfe00a] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.

[Pfe00b] Frank Pfenning. Structural cut elimination: I. Intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[Pfe01] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, MA, June 2001. IEEE Computer Society Press.

[Pie00] Brigitte Pientka. Termination and reduction checking in the logical framework. In *Workshop on Automation of Proofs by Mathematical Induction*, Pittsburgh, Pennsylvania, June 2000.

[PP03] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. Submitted for publication, February 2003.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

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

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.

[SP03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. Submitted for publication, February 2003.

[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.