A Linear Logical Framework

Iliano Cervesato and Frank Pfenning*
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
{*iliano*|*fp*}*@cs.cmu.edu*

Abstract

We present the linear type theory LLF as the formal basis for a conservative extension of the LF logical framework. LLF combines the expressive power of dependent types with linear logic to permit the natural and concise representation of a whole new class of deductive systems, namely those dealing with state. As an example we encode a version of Mini-ML with references including its type system, its operational semantics, and a proof of type preservation. Another example is the encoding of a sequent calculus for classical linear logic and its cut elimination theorem. LLF can also be given an operational interpretation as a logic programming language under which the representations above can be used for type inference, evaluation and cut-elimination.

1 Introduction

A logical framework is a formal system designed to provide effective representations of deductive systems and their properties. Proposals based on intuitionistic logic and intuitionistic type theory, such as the logical framework *LF* [9], have been widely used to study logical formalisms [19] and programming languages [11]. Unfortunately, many constructs and concepts needed in common programming practice cannot be represented in a satisfactory way in these meta-languages. In particular, constructs based on the notion of state as found in imperative languages often escape an elegant formalization by means of these tools. Similarly, logical systems that, by definition (e.g. substructural logics) or by presentation (e.g. Dyckhoff's contraction-free intuitionistic sequent calculus [5]), rely on destructive context reductions require awkward encodings in an intuitionistic framework. Consequently the adequacy of the representation is difficult to prove and the formal meta-theory quickly becomes intractable.

Linear logic [7] provides a view of context formulas as resources, which can be exploited to model the notion of state, as described for example in [4, 10, 12, 22]. The current proposals put the emphasis on the issue of *representing* imperative constructs and resource-based logics, but appear inadequate for *reasoning* effectively about these representations. On the other hand, intuitionistic type-theoretic frameworks such as *LF* make the representation of meta-reasoning easy, but do not have any notion of linearity built in.

As a concrete example, consider the problem of representing a cut-elimination procedure for linear logic. Given the derivation $\mathcal{D}$ for a linear sequent $\Delta \longrightarrow \Theta$, this procedure produces an equivalent cut-free derivation $\mathcal{D}'$. Since it operates linearly on the formulas appearing in it, $\mathcal{D}$ would be adequately represented by a term $\ulcorner\mathcal{D}\urcorner$ in a *linear* λ-calculus; the same holds for $\mathcal{D}'$. This problem was encoded in *LF* by representing sequents as types and derivations as proof terms [18]. *LF* is intuitionistic and therefore the linearity of $\ulcorner\mathcal{D}\urcorner$ needed to be checked explicitly as a property of $\ulcorner\mathcal{D}\urcorner$, complicating the meta-theory to the extent that it became infeasible and only the cut-elimination algorithm without the linearity check was implemented. On the other hand, encoding the same problem in Miller's linear meta-logic *Forum* [12] would map linear sequent derivations to linear derivations in the meta-language. But since *Forum* lacks proof terms, no internal notation for those entities is provided and cut-elimination cannot be implemented in this manner.

In this paper, we propose a conservative extension of the logical framework *LF* that permits representing linear objects and reasoning about them. This formalism, that we call *Linear LF*, or *LLF* for short, is a type theory featuring linear function types ($\multimap$), additive pairing (&), unit type ($\top$), and intuitionistic dependent function types (Π). *LLF* extends the ob-

*This work was supported by NSF Grant CCR-9303383. The second author was supported by the *Alexander-von-Humboldt-Stiftung* when this paper was completed, during a visit to the Department of Mathematics of the Technical University Darmstadt.

This extended abstract will appear on the proceedings of the *Eleventh Annual IEEE Symposium on Logic in Computer Science — LICS'96*, New Brunswick, NJ, July 27–30 1996.

jects of LF with linear functional abstraction, additive pairs and unit, the corresponding destructors, and their equational theory. In order to keep the system simple we restrict the indices of type families to be linearly closed so that a type can depend only on intuitionistic assumptions, but not on linear variables. While at first this may appear to be a strong restriction, the resulting system is surprisingly expressive, allowing the faithful implementation of cut elimination for classical linear logic, translations between linear natural deduction and sequent calculus, and properties of imperative languages such as type preservation.

LLF also maintains the computational nature of LF and is amenable to an efficient implementation as a logic programming language in the style of *Elf* [15, 17]. The experience gained with linearity in the language *Lolli* [10, 2] applies directly to our formalism.

The principal contributions of this work are: (1) the definition of a uniform type theory admitting linear entities in conjunction with dependent types; (2) the use of this system as a logical framework to represent and reason about problems that are not handled well by previous formalisms, either linear or intuitionistic; (3) the description of an operational system which constitutes the first step towards an implementation of this formalism as a linear constraint logic programming language. To our knowledge, this is the first linear type theory that goes beyond simple types. Our work was inspired by ideas in [14].

The paper is organized as follows. Section 2 describes LLF and presents major results in its metatheory, such as the decidability of type-checking. Section 3 indicates that efficient proof search in the style of logic programming can be achieved in LLF. Section 4 illustrates the expressiveness of LLF on an example featuring imperative computations. Section 5 presents cut-elimination for classical linear logic as another example. Finally, Section 6 assesses the results and outlines future work.

2 The Linear Logical Framework *LLF*

In order to facilitate the description of LLF in the available space, we must assume that the reader is familiar with both the logical framework LF [9] and various presentations of linear logic [7, 8]. We will also sparingly take advantage of the natural extension of the Curry-Howard isomorphism to linear logic by viewing types as formulas.

LLF extends the logical framework LF with three connectives from linear logic, seen in this context as type constructors, namely *multiplicative implication* ($\multimap$), *additive conjunction* ($\&$) and *additive truth* ($\top$). The language of objects is augmented accordingly with the respective constructors and destructors. Linear types operate on *linear assumptions* which we represent as distinguished declarations of the form $x \hat{:} A$ in the context; we write $x : A$ for context elements à la LF and call them *intuitionistic assumptions*. The syntax of LLF is given by the following grammar (constructs not present in LF are separated by a double bar $\|$):

$$
\begin{array}{rrcl}
\textit{Objects:} & M & ::= & c \mid x \mid \lambda x : A.\, M \mid M_1\, M_2 \\
 & & & \| \ \langle\rangle \mid \langle M_1, M_2 \rangle \mid \textsc{fst}\ M \mid \textsc{snd}\ M \\
 & & & \mid \hat{\lambda} x : A.\, M \mid M_1 \,\hat{}\, M_2 \\
\textit{Families:} & P & ::= & a \mid P\, M \\
\textit{Types:} & A & ::= & P \mid \Pi x : A_1.\, A_2 \\
 & & & \| \ \top \mid A_1 \,\&\, A_2 \mid A_1 \multimap A_2 \\
\textit{Kinds:} & K & ::= & \textsc{type} \mid \Pi x : A.\, K \\
\textit{Contexts:} & \Psi & ::= & \cdot \mid \Psi, x : A \ \| \ \Psi, x \hat{:} A \\
\textit{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma, a : K \mid \Sigma, c : A
\end{array}
$$

Here x, c and a range over object-level variables, object constants and type family constants, respectively. In addition to the names displayed above, we will often use N and B to range over objects and types respectively. Moreover, we denote generic terms, i.e. objects, types or kinds, with the letters U and V, possibly subscripted. As usual, we write $A \to U$ for $\Pi x : A.\, U$ whenever x does not occur in the type or kind U.

The notions of free and bound variables are adapted from LF (notice the presence of a new binding construct: linear λ-abstraction). As usual, we identify terms that differ only in the name of their bound variables and write $[M/x]U$ for the capture-avoiding substitution of M for x in the term U. Finally we require variables and constants to be declared at most once in a context and in a signature, respectively.

Below we will often need to access the *intuitionistic part* of a context. Therefore, we introduce the function $\overline{\Psi}$ defined as follows:

$$
\begin{cases}
\overline{\cdot} & = \ \cdot \\
\overline{\Psi, x : A} & = \ \overline{\Psi}, x : A \\
\overline{\Psi, x \hat{:} A} & = \ \overline{\Psi}
\end{cases}
$$

We overload this notation and use $\overline{\Psi}$ to express the fact that the linear portion of the denoted context is constrained to be empty (e.g. in the all rules for type families in Figure 1).

The meaning of the syntactic entities of LLF can be presented in various forms. Figures 1 and 2 show a version of the type system for LLF that we call *pre-canonical* and that is particularly well-suited for proving meta-theoretic properties about this formalism. The system relies on the following seven judgments:

$\vdash \Sigma \Uparrow \mathit{Sig}$ (Σ *is a pre-canonical signature*)
$\vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$ (Ψ *is a pre-canonical context*)

Signatures

$$\frac{}{\vdash \cdot \Uparrow \mathit{Sig}}\ \textbf{s\_dot} \qquad \frac{\vdash \Sigma \Uparrow \mathit{Sig} \quad \cdot \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash \Sigma, c\!:\!A \Uparrow \mathit{Sig}}\ \textbf{s\_obj} \qquad \frac{\vdash \Sigma \Uparrow \mathit{Sig} \quad \cdot \vdash_\Sigma K \Uparrow \mathit{Kind}}{\vdash \Sigma, a\!:\!K \Uparrow \mathit{Sig}}\ \textbf{s\_fam}$$

Contexts

$$\frac{\vdash \Sigma \Uparrow \mathit{Sig}}{\vdash_\Sigma \cdot \Uparrow \mathit{Ctx}}\ \textbf{c\_dot} \qquad \frac{\vdash_\Sigma \Psi \Uparrow \mathit{Ctx} \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash_\Sigma \Psi, x\!:\!A \Uparrow \mathit{Ctx}}\ \textbf{c\_int} \qquad \frac{\vdash_\Sigma \Psi \Uparrow \mathit{Ctx} \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash_\Sigma \Psi, x \hat{:} A \Uparrow \mathit{Ctx}}\ \textbf{c\_lin}$$

Kinds

$$\frac{\vdash_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash_\Sigma \text{TYPE} \Uparrow \mathit{Kind}}\ \textbf{kc\_type} \qquad \frac{\overline{\Psi}, x\!:\!A \vdash_\Sigma K \Uparrow \mathit{Kind}}{\overline{\Psi} \vdash_\Sigma \Pi x\!:\!A.\, K \Uparrow \mathit{Kind}}\ \textbf{kc\_dep}$$

Types/type families

$$\frac{\overline{\Psi} \vdash_\Sigma P \downarrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma P \Uparrow \text{TYPE}}\ \textbf{fc\_a} \qquad \frac{\vdash_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash_\Sigma \top \Uparrow \text{TYPE}}\ \textbf{fc\_top} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma A \,\&\, B \Uparrow \text{TYPE}}\ \textbf{fc\_with}$$

$$\frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma A \multimap B \Uparrow \text{TYPE}}\ \textbf{fc\_limp} \qquad \frac{\overline{\Psi}, x\!:\!A \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma \Pi x\!:\!A.\, B \Uparrow \text{TYPE}}\ \textbf{fc\_dep}$$

(No **fc_eq**, no **fa_c**)

$$\frac{\overline{\Psi} \vdash_\Sigma A \downarrow K \quad K \equiv K' \quad \overline{\Psi} \vdash_\Sigma K' \Uparrow \mathit{Kind}}{\overline{\Psi} \vdash_\Sigma A \downarrow K'}\ \textbf{fa\_eq}$$

$$\frac{\vdash_{\Sigma, a:K, \Sigma'} \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash_{\Sigma, a:K, \Sigma'} a \downarrow K}\ \textbf{fa\_con} \qquad \frac{\overline{\Psi} \vdash_\Sigma P \downarrow \Pi x\!:\!A.\, K \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\overline{\Psi} \vdash_\Sigma P\, N \downarrow [N/x]K}\ \textbf{fa\_iapp}$$

Figure 1. A Pre-canonical Deduction System for LLF, kinds and types

$\overline{\Psi} \vdash_\Sigma K \Uparrow \mathit{Kind}$ (K *is a pre-canonical kind*)
$\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}$ (A *is a pre-canonical type*)
$\overline{\Psi} \vdash_\Sigma A \downarrow K$ (A *is a pre-atomic family of kind* K)
$\Psi \vdash_\Sigma M \Uparrow A$ (M *is a pre-canonical obj. of type* A)
$\Psi \vdash_\Sigma M \downarrow A$ (M *is a pre-atomic object of type* A)

plus the three judgments for definitional equality

$U \equiv V$ (U *is definitionally equal to* V)

and a judgment for non-deterministically splitting (or merging, depending on the point of view) the context

$\Psi = \Psi' \bowtie \Psi''$ (Ψ *splits into* Ψ' *and* Ψ'').

Whenever a property holds uniformly for the five judgments concerning objects, types and kinds above, we take the liberty of writing $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$ and then referring to the generic terms U and V.

A few remarks on these judgments are in order prior to describing the rules in Figures 1–2. The notion of definitional equality that we consider is the equivalence constructed on the congruence relation built out of the following β-reduction rules:

$$\begin{aligned}
\beta_{fst}: &\quad \text{FST}\,\langle M, N\rangle \longrightarrow M \\
\beta_{snd}: &\quad \text{SND}\,\langle M, N\rangle \longrightarrow N \\
\beta_{lapp}: &\quad (\hat{\lambda} x\!:\!A.\, M)\,\hat{}\,N \longrightarrow [N/x]M \\
\beta_{iapp}: &\quad (\lambda x\!:\!A.\, M)\, N \longrightarrow [N/x]M.
\end{aligned}$$

The omission of the rules for η-expansion is justified by the fact that the interplay of the rules for pre-canonical and pre-atomic objects in Figure 2 guarantees that a pre-canonical object is always in η-long form. Moreover, it is easy to show that this property is hereditarily maintained when substituting a pre-canonical object in a pre-canonical type or kind, and by β-reduction. Therefore, all terms satisfying the judgments above are in η-long form, although they might contain β-redices, and therefore are not necessarily in normal form. Distinguishing two object-level forms of judgments instead of just defining one typing judgment is essential in order to achieve this property. It has the practical effects of avoiding the meta-theoretic complications introduced by explicitly considering the η-conversion rules. Moreover, it lays the basis for an efficient implementation strategy in which η-expansion takes place in a preprocessing phase but is not required during the execution. Normalization is then simply β-reduction. Instead, a direct extension of the current implementation techniques for *LF* [17] would require carrying types around in order to handle properly objects of type $\top$. To our knowledge, this is the first formulation of a type theory that focuses uniquely on long forms. It was inspired by Felty's *canonical LF* [6].

Traditional presentations of linear logic define the

Context splitting/merging

$$\frac{}{\cdot = \cdot \bowtie \cdot}\ \textbf{s\_dot} \qquad \frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x \mathbin{\hat{:}} A) = (\Psi', x \mathbin{\hat{:}} A) \bowtie \Psi''}\ \textbf{s\_lin1}$$

$$\frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x : A) = (\Psi', x : A) \bowtie (\Psi'', x : A)}\ \textbf{s\_int} \qquad \frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x \mathbin{\hat{:}} A) = \Psi' \bowtie (\Psi'', x \mathbin{\hat{:}} A)}\ \textbf{s\_lin2}$$

Objects

$$\frac{\Psi \vdash_\Sigma M \downarrow P}{\Psi \vdash_\Sigma M \Uparrow P}\ \textbf{oc\_a} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow B \quad A \equiv B \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\Psi \vdash_\Sigma M \Uparrow A}\ \textbf{oc\_eq}$$

$$\frac{\vdash_\Sigma \Psi \Uparrow Ctx}{\Psi \vdash_\Sigma \langle\rangle \Uparrow \top}\ \textbf{oc\_unit} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow A \quad \Psi \vdash_\Sigma N \Uparrow B}{\Psi \vdash_\Sigma \langle M, N\rangle \Uparrow A \,\&\, B}\ \textbf{oc\_pair}$$

$$\frac{\Psi, x \mathbin{\hat{:}} A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \hat{\lambda} x : A.\, M \Uparrow A \multimap B}\ \textbf{oc\_llam} \qquad \frac{\Psi, x : A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \lambda x : A.\, M \Uparrow \Pi x : A.\, B}\ \textbf{oc\_ilam}$$

$$\frac{\Psi \vdash_\Sigma M \Uparrow A}{\Psi \vdash_\Sigma M \downarrow A}\ \textbf{oa\_c} \qquad \frac{\Psi \vdash_\Sigma M \downarrow B \quad A \equiv B \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\Psi \vdash_\Sigma M \downarrow A}\ \textbf{oa\_eq}$$

$$\frac{\vdash_{\Sigma, c:A, \Sigma'} \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash_{\Sigma, c:A, \Sigma'} c \downarrow A}\ \textbf{oa\_con} \qquad \frac{\vdash_\Sigma \overline{\Psi}, x \mathbin{\hat{:}} A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x \mathbin{\hat{:}} A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{oa\_lvar} \qquad \frac{\vdash_\Sigma \overline{\Psi}, x : A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x : A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{oa\_ivar}$$

(No rule for $\top$)

$$\frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \mathrm{FST}\, M \downarrow A}\ \textbf{oa\_fst} \qquad \frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \mathrm{SND}\, M \downarrow B}\ \textbf{oa\_snd}$$

$$\frac{\Psi' \vdash_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash_\Sigma M \hat{}\, N \downarrow B}\ \textbf{oa\_lapp} \qquad \frac{\Psi \vdash_\Sigma M \downarrow \Pi x : A.\, B \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\Psi \vdash_\Sigma M\, N \downarrow [N/x]B}\ \textbf{oa\_iapp}$$

Figure 2. A Pre-canonical Deduction System for LLF, objects

context Δ of an intuitionistic sequent $\Delta \longrightarrow A$ as a multiset of linear formulas. More recent works [8, 10, 18] prefer to refine it as a pair $\Gamma; \Delta$ of multisets so that the sequent $\Gamma; \Delta \longrightarrow A$ is equi-provable with $(!\Gamma, \Delta) \longrightarrow A$ in the traditional formulation. The two components of $\Gamma; \Delta$ are called the intuitionistic and the linear context, respectively. Our presentation adopts the latter strategy by distinguishing intuitionistic $(x : A)$ and linear $(x \mathbin{\hat{:}} A)$ variable declarations. However, in *LLF* as in *LF*, an assumption can depend on previous declarations; their order is therefore critical and we must adopt sequences as the representation for contexts.

The constraints on the structure of an *LLF* context lead to small notational complications. The axiom rules of linear logic, which close a proof tree, are applicable only if the linear context is empty, or if it only consists of the formula to be proved. The corresponding *LLF* rules (**oa_con**, **oa_lvar** and **oa_ivar**) rely on the notation $\overline{\cdots}$ to express the same conditions. Another complication arises when it comes to representing context splitting for multiplicative connectives. Lacking the commutativity properties of multisets, we use a separate judgment in rule **oa_lapp** to specify that the linear context in the conclusion should be split among the premisses of this rule. The rules in the upper part of Figure 2 define it.

The rules concerning linear objects in Figure 2 define the behavior of linear types. If we ignore the objects and the distinction between the two judgments, they correspond to the specification of the familiar rules for the linear connectives $\top$, & and $\multimap$, presented in a *natural deduction* style. It is easy to prove the equivalence to the usual sequent formulation. The objects that appear on the left of these types record the structure of a natural deduction proof for the corresponding linear formulas.

The dependent function type $\Pi x : A.\, B$ that *LLF* inherits from *LF* generalizes both intuitionistic implication $A \to B$ (customarily defined as $!A \multimap B$) and the universal quantifier $\forall x.\, B$, where A plays the role of the type of the (intuitionistic) variable x.

With the interpretation above, *LLF* makes available all the connectives and quantifiers of the freely generated fragment of the language of linear hereditary Harrop formulas, on which the programming language *Lolli* is based [10]. Additionally, *LLF* offers the characteristic features of a type theory: higher-order functions, proof terms, and type families indexed by arbitrary objects, possibly higher-order and linear.

In the rules in Figures 1–2, types and kinds are always checked using a purely intuitionistic context. This prevents valid types from containing free linear variables. Loosening this restriction would require admitting linear dependent function types in our language, corresponding to linear quantifiers. Although this inclusion appears beneficial for certain applications, preliminary investigations seem to indicate that the consequent complications might outweigh the potential advantages.

LLF enjoys the same meta-theoretic properties that hold for *LF*. The principal results are summarized as follows:

Theorem

1. (Church-Rosser property) *Whenever* $U \equiv U'$, *there is a term* V *such that* $U \longrightarrow^* V$ *and* $U' \longrightarrow^* V$, *where* $\longrightarrow^*$ *is the transitive closure of the reduction congruence* $\longrightarrow$.
2. (Unicity of types and kinds) *If* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *and* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V'$ *are both derivable, then* $V \equiv V'$.
3. (Strong normalization) *If* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *is derivable, then* U *is strongly normalizing.*
4. (Decidability of type checking) *It can be recursively decided whether there exist derivations for the judgments* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, $\vdash_\Sigma \Psi \Uparrow Ctx$ *and* $\vdash \Sigma \Uparrow Sig$.
5. (Conservativity over *LF*) *If* Σ, Ψ, U *and* V *do not mention linear constructs, then* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *is derivable* iff $\Psi \vdash_\Sigma^{\mathrm{LF}} U \Uparrow\!\!\Downarrow V$ *is derivable in the* LF *type theory, and similarly for contexts and signatures.* □

The proofs of these results adapt the techniques outlined in [9] for *LF*. However, differently from that treatment, our ability to work uniquely with η-long forms permits obtaining precisely the canonical terms that are needed for meta-representation and proof-search. The reader is invited to consult [1] for details.

Since type checking for *LLF* is decidable, we can effectively determine if a given term is the representation of a valid derivation from a (potentially linear) object logic, which is crucial in applications of a logical framework. Being a conservative extension over *LF* is important, since all representation techniques, adequacy theorems, and examples developed for *LF* remain valid for *LLF*. This also means that, under the computational interpretation of section 3, any *LF* program is a valid *LLF* program and will behave in exactly the same way.

From the explanation above, it should be clear that the normal form of a pre-canonical term is indeed canonical (that is, in long $\beta\eta$-normal form). A canonical proof system for *LLF* can easily be obtained from the calculus in Figures 1–2: we replace the equivalence rules **fa_eq**, **oc_eq** and **oa_eq** with applications of the *normalization function* NF(_) in the rules involving substitution. NF(_) exists by virtue of strong normalization; moreover, by removing the rule **oa_c**, we guarantee that derivable objects do not contain β-redices, and are therefore canonical. We write the canonical derivability judgments in the form $\Psi \vdash_\Sigma U \Uparrow V$ (notice the different arrow shape).

3 Logic Programming with *LLF*

The canonical system outlined at the end of the last section is adequate for checking that an object has a given type, but not for generating a (canonical) term of that type. Similarly to the case of *LF*, this problem, proof-search under the Curry-Howard isomorphism, can be efficiently mechanized in *LLF*. This lays the basis for converting our formalism into a linear constraint logic programming language.

For the purpose of motivating this statement, let us cast our system into the usual terminology of logic programming. We use 'formula' instead of 'type' and, given the judgment $\Psi \vdash_\Sigma M \Uparrow A$, we call A its goal and adopt the name 'program' for the combined context Σ, Ψ (overloading ',' to denote concatenation). We aim at finding a derivation, encoded as M, of the goal A from the program Σ, Ψ. For this purpose, we want to be able to interpret the connectives (type constructors in our context) of A as search directives and the declarations in Σ, Ψ as partial definitions for the atomic formulas that might appear in the goal, so that program formulas need to be accessed only when the goal is atomic. A proof is goal-oriented if the program is accessed only after the goal has been reduced to an atomic formula. A proof is focused if every time a program formula is considered, it can be processed up to the atoms it defines without needing to access any other program formula. A proof having both these properties is uniform, and a formalism such that every provable goal has a uniform proof is called an abstract logic programming language [13].

LLF is an abstract logic programming language and Figure 3 describes a uniform proof system for it. The rules for the uniform provability judgment $\Psi \xrightarrow{u}_\Sigma M : A$, in the upper part, process the goal up to an atomic

Uniform provability

$$\frac{\Psi \xrightarrow{u}_{\Sigma,c:A,\Sigma'} c : A \gg M : P}{\Psi \xrightarrow{u}_{\Sigma,c:A,\Sigma'} M : P}\ \textbf{ou\_con} \qquad \frac{\Psi, x{:}A, \Psi' \xrightarrow{u}_{\Sigma} x : A \gg M : P}{\Psi, x{:}A, \Psi' \xrightarrow{u}_{\Sigma} M : P}\ \textbf{ou\_ivar} \qquad \frac{\Psi, \Psi' \xrightarrow{u}_{\Sigma} x : A \gg M : P}{\Psi, x\hat{:}A, \Psi' \xrightarrow{u}_{\Sigma} M : P}\ \textbf{ou\_lvar}$$

$$\frac{}{\Psi \xrightarrow{u}_{\Sigma} \langle\rangle : \top}\ \textbf{ou\_unit} \qquad \frac{\Psi \xrightarrow{u}_{\Sigma} M : A \quad \Psi \xrightarrow{u}_{\Sigma} N : B}{\Psi \xrightarrow{u}_{\Sigma} \langle M, N\rangle : A \,\&\, B}\ \textbf{ou\_pair}$$

$$\frac{\overline{\Psi} \vdash_{\Sigma} A \Uparrow \mathrm{TYPE} \quad \Psi, x\hat{:}A \xrightarrow{u}_{\Sigma} M : B}{\Psi \xrightarrow{u}_{\Sigma} \hat{\lambda} x{:}A.\, M : A \multimap B}\ \textbf{ou\_llam} \qquad \frac{\overline{\Psi} \vdash_{\Sigma} A \Uparrow \mathrm{TYPE} \quad \Psi, x{:}A \xrightarrow{u}_{\Sigma} M : B}{\Psi \xrightarrow{u}_{\Sigma} \lambda x{:}A.\, M : \Pi x{:}A.\, B}\ \textbf{ou\_ilam}$$

Immediate entailment

$$\frac{}{\overline{\Psi} \xrightarrow{u}_{\Sigma} M : P \gg M : P}\ \textbf{oi\_atm}$$

$$\frac{\Psi \xrightarrow{u}_{\Sigma} \mathrm{FST}\, M : A \gg N : P}{\Psi \xrightarrow{u}_{\Sigma} M : A \,\&\, B \gg N : P}\ \textbf{oi\_fst} \qquad \frac{\Psi \xrightarrow{u}_{\Sigma} \mathrm{SND}\, M : B \gg N : P}{\Psi \xrightarrow{u}_{\Sigma} M : A \,\&\, B \gg N : P}\ \textbf{oi\_snd}$$

$$\frac{\Psi'' \xrightarrow{u}_{\Sigma} M\,\hat{}\,M' : B \gg N : P \quad \Psi' \xrightarrow{u}_{\Sigma} M' : A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \xrightarrow{u}_{\Sigma} M : A \multimap B \gg N : P}\ \textbf{oi\_lapp}$$

$$\frac{\Psi \xrightarrow{u}_{\Sigma} M\, M' : \mathrm{NF}([M'/x]B) \gg N : P \quad \overline{\Psi} \vdash_{\Sigma} M' \Uparrow A}{\Psi \xrightarrow{u}_{\Sigma} M : \Pi x{:}A.\, B \gg N : P}\ \textbf{oi\_iapp\_static}$$

$$\frac{\Psi \xrightarrow{u}_{\Sigma} M\, M' : \mathrm{NF}([M'/x]B) \gg N : P \quad \overline{\Psi} \xrightarrow{u}_{\Sigma} M' : A}{\Psi \xrightarrow{u}_{\Sigma} M : \Pi x{:}A.\, B \gg N : P}\ \textbf{oi\_iapp\_dynamic}$$

Figure 3. A Uniform Deduction System for LLF

formula, then select an assumption from the program (rules **ou_con**, **ou_lvar** and **ou_ivar**) and pass it to the immediate entailment judgment $\Psi \xrightarrow{u}_{\Sigma} M : A \gg N : P$ that decomposes it until it matches the goal (rule **oi_atm**). The term A in the program formula $\Pi x{:}A.\, B$ can be viewed either as the type of the term M' to which the variable x should be instantiated (rule **oi_iapp_static**), or as a formula to be proved by proof-search (rule **oi_iapp_dynamic**); a complete discussion on this aspect can be found in [1]. Notice that types in Figure 3 are checked for being canonical in rules **ou_llam**, **ou_ilam** and **oi_iapp_static**, therefore this deduction system does not replace, but extends the full canonical system.

This system is sound and complete with respect to the canonical calculus outlined at the end of Section 2.

Theorem $\Psi \xrightarrow{u}_{\Sigma} M : A$ *iff* $\Psi \vdash_{\Sigma} M \Uparrow A$

Proof: After a proper generalization to handle atomic derivability and immediate entailment, we proceed by structural induction on the derivations in the hypotheses. See [1] for details. □

We should stress that this is just the first step towards the specification of *LLF* as a logic programming language. In particular, a number of non-deterministic choices remain implicit in Figure 3. Specifically, no strategy is provided to split the context in rule **oi_lapp** (resource management non-determinism); no execution order is specified for the premisses of rule **oi_lapp** and **oi_iapp** (conjunctive non-determinism); no criterion is given in order to pick a program formula when a goal is atomic in rules **ou_con**, **ou_lvar** and **ou_ivar** (disjunctive non-determinism); and finally, no recipe is given to choose the instantiating object M' in rule **oi_iapp_static** (existential non-determinism). We expect to be able to adapt standard techniques [16] to our linear setting in order to handle these issues effectively.

4 An Example: Reasoning about Imperative Computations

In [4] Chirimar demonstrates how *Forum* [12], a meta-logic based on classical linear logic, can serve as a natural meta-language for specifying *UML*, a version of *Mini-ML* with references and continuations. In

$$
\begin{array}{llll}
(\textit{References}) & \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}\, e : \tau\ \mathrm{ref}}\ \textsf{ofe\_ref} & \dfrac{\Gamma \vdash e : \tau\ \mathrm{ref}}{\Gamma \vdash\ !\, e : \tau}\ \textsf{ofe\_deref} & \dfrac{\Gamma \vdash e_1 : \tau\ \mathrm{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}\ \textsf{ofe\_assign} \\[2ex]
(\textit{Definitions}) & \dfrac{\Gamma \vdash [v/x]e : \tau}{\Gamma \vdash \mathbf{letv}\ x = v\ \mathbf{in}\ e : \tau}\ \textsf{ofe\_letv} & & \dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \textsf{ofe\_let} \\[2ex]
(\textit{Cells}) & \dfrac{}{\Delta \vdash \cdot : \cdot}\ \textsf{ofc\_empty} & & \dfrac{\Delta \vdash S : \Delta' \quad \Delta \vdash v : \tau}{\Delta \vdash (S, c = v) : (\Delta', c{:}\tau)}\ \textsf{ofc\_cell} \\[2ex]
(\textit{Answers}) & \dfrac{\Delta \vdash S : \Delta \quad \Delta \vdash v : \tau}{\Delta \vdash (S, v) : \tau}\ \textsf{off\_loc} & & \dfrac{\Delta, c{:}\tau' \vdash w : \tau}{\Delta \vdash \mathbf{new}\ c.\ w : \tau}\ \textsf{off\_new}
\end{array}
$$

Figure 4. Some Typing Rules for MLR

this section we sketch a similar (and executable) specification of *Mini-ML* with references (ML^{ref}) in *LLF* and then show how to implement the proof of *type preservation* for ML^{ref}. Among other things, our encoding indicates that the classical operators of *Forum* are not essential in this example; this observation extends to a broad class of applications in the theory of programming languages. It furthermore demonstrates how the combination of linear and dependently typed terms can be employed to mechanize the meta-theory of languages with state at a very high level of abstraction which is not achievable in other frameworks.

The complete *LLF* code for this example can be found in [3]. There and in this section, we adopt a concrete syntax analogous to that of *Elf* [17]. In particular, we use `{x:A}B` for $\Pi x : A.\, B$, simplifying it to `A -> B` or `B <- A` whenever possible. We adopt `-o` or `o-` for $\multimap$, `&` for & and `<T>` for $\top$. We denote the objects $\langle\rangle$, $\langle M, N\rangle$, $\lambda x : A.\, M$, $\hat{\lambda} x : A.\, M$ and $M\,\hat{}\,N$ as `()`, `(M,N)`, `[x:A]M`, `[x^A]M` and `M^N`, respectively. Whenever possible, we keep the types implicit in the binding constructs.

The polymorphism in ML^{ref} is restricted to values [21] which seems to be generally accepted as superior to *SML*'s imperative type variables. We achieve this by distinguishing two forms of **let**. We use x to stand for variables and c to stand for addresses of cells which may be updated imperatively. In the informal presentation we think of values as a subset of all possible expressions and overload constructors between values and expressions as usual. The implementation distinguishes them as different *LLF* types. We elide the standard expression and value constructors and destructors for natural numbers, unit, pairs, functions and recursion.

$$
\begin{array}{rrl}
\textit{Types} & \tau & ::= \mathrm{nat} \mid 1 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \tau\ \mathrm{ref} \\
\textit{Expressions} & e & ::= v \mid \ldots \mid \mathbf{ref}\, e \mid\ !\, e \mid e_1 := e_2 \\
 & & \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{letv}\ x = v\ \mathbf{in}\ e
\end{array}
$$

$$
\begin{array}{rrl}
\textit{Values} & v & ::= \ldots \mid x \mid \mathbf{ref}\, c \\
\textit{States} & S & ::= \cdot \mid S, c = v \\
\textit{Answers} & w & ::= (S, v) \mid \mathbf{new}\ c.\ w
\end{array}
$$

To express the basic judgments we also need *continuations* and *contexts*. *Contexts* declare types for variables and reference cells, *store contexts* only for cells.

$$
\begin{array}{rrcl}
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \mid \Gamma, c{:}\tau \\
\textit{Store contexts} & \Delta & ::= & \cdot \mid \Delta, c{:}\tau \\
\textit{Continuations} & K & ::= & \mathbf{init} \mid K; \lambda x.\ e
\end{array}
$$

A continuation is either initial or constructed by adding an instruction to a continuation. We can think of continuations functionally (in which case **init** is the identity and ";" is function composition) or as a stack of instructions. ML^{ref} is represented using standard techniques of higher-order abstract syntax; no linearity is needed at this level. An important point about the representation is that the spaces of expressions `exp`, values `val`, cells `cell`, answers `final`, and continuations `cont` are all separate types, with some explicit coercions between them. This is necessary since the framework lacks subtyping. Note also that there are no constructors of type `cell` in the signature: they are only introduced as parameters during the execution of a program when storage cells are allocated.

We have four typing judgments:

$$
\begin{array}{ll}
\Gamma \vdash e : \tau & (e\ \textit{has type}\ \tau) \\
\Gamma \vdash K : \tau_1 \Rightarrow \tau_2 & (K\ \textit{maps values in}\ \tau_1\ \textit{to answers in}\ \tau_2) \\
\Delta \vdash S : \Delta' & (\textit{Cells of}\ S\ \textit{have types prescribed in}\ \Delta') \\
\Delta \vdash w : \tau & (\textit{Answer}\ w\ \textit{has type}\ \tau)
\end{array}
$$

respectively represented, following the usual judgments-as-types methodology of *LF*, as the type families

```
ofe: exp -> tp -> type.
ofk: cont -> tp -> tp -> type.
ofc: cell -> tp -> type.
off: final -> tp -> type.
```

$$\frac{S \triangleright K; \lambda x.\, \mathbf{ref}\, x \vdash e \hookrightarrow w}{S \triangleright K \vdash \mathbf{ref}\, e \hookrightarrow w}\,\text{ex\_ref} \qquad \frac{S, c = v \triangleright K \vdash \mathbf{ref}\, c \hookrightarrow w}{S \triangleright K \vdash \mathbf{ref}\, v \hookrightarrow \mathbf{new}\, c.\, w}\,\text{ex\_ref1}$$

$$\frac{S', c = v, S'' \triangleright K \vdash v \hookrightarrow w}{S', c = v, S'' \triangleright K \vdash \mathbf{!}\,(\mathbf{ref}\, c) \hookrightarrow w}\,\text{ex\_deref1} \qquad \frac{S', c = v_2, S'' \triangleright K \vdash \langle\,\rangle \hookrightarrow w}{S', c = v_1, S'' \triangleright K \vdash \mathbf{ref}\, c := v_2 \hookrightarrow w}\,\text{ex\_assign2}$$

$$\frac{S \triangleright K \vdash [v/x]e \hookrightarrow w}{S \triangleright K \vdash \mathbf{letv}\, x = v\, \mathbf{in}\, e \hookrightarrow w}\,\text{ex\_letv} \qquad \frac{S \triangleright K \vdash [v/x]e \hookrightarrow w}{S \triangleright K; \lambda x.\, e \vdash v \hookrightarrow w}\,\text{ex\_return} \qquad \frac{}{S \triangleright \mathbf{init} \vdash v \hookrightarrow (S, v)}\,\text{ex\_init}$$

Figure 5. Some Evaluation Rules for MLR

The type of the store is represented in a distributed fashion with a separate typing assumptions for each cell. We show some of the critical rules defining these judgments in Figure 4.

In the rule for **letv** it may be desirable to also check that v has at least some type, but this is not necessary to prove type preservation. In practice, one would of course implement this rule by computing a principal type scheme for v, but we consider this an optimization which is orthogonal to our declarative system. The sample declarations below implement the first four inference rules in Figure 4. Free variables in a declaration are implicitly Π-quantified with a type which is determined during *LLF* type reconstruction.

```
ofe_ref: ofe (ref E) (rf T)
          <- ofe E T.
ofe_deref: ofe (deref E) T
          <- ofe E (rf T).
ofe_assign: ofe (assign E1 E2) 1
          <- ofe E1 (rf T)
          <- ofe E2 T.
ofe_letv: ofe (letv V ([x:val] E x)) T
          <- ofe (E V) T.
```

The continuation-based operational semantics evaluates an expression e given a state S and a continuation K to an answer w which encapsulates the final state. We write $S \triangleright K \vdash e \hookrightarrow w$. We show some of the critical rules concerned with state in Figure 5. Recall that **ref** v is not a value, but that **ref** c is. The last two rules show that values are returned by applying the continuation or constructing the final answer.

In rule **ex_ref1**, the cell c must be new, that is, not occur in S, K, or v. Note that the evaluation rule for the polymorphic let, **letv**, matches the typing rule, which is critical in the proof of the type preservation theorem below.

The representation of the evaluation judgment once again follows the judgments-as-types methodology, but we treat the state S specially as will be discussed below. A judgment $S \triangleright K \vdash e \hookrightarrow w$ is encoded as the type `exec` $\ulcorner K \urcorner$ (`ev` $\ulcorner e \urcorner$) $\ulcorner w \urcorner$, where `ev` is merely a coercion from expressions `exp` to machine instructions `inst` which also encompass values v, which are coerced as (`return` $\ulcorner v \urcorner$). The state S is represented by an *LLF* context with a *linear* hypothesis for every cell in S:

$$\begin{aligned} \ulcorner \cdot \urcorner &= \cdot \\ \ulcorner S, c = v \urcorner &= \ulcorner S \urcorner,\ c{:}\mathtt{cell},\ c' \,\hat{:}\, \mathtt{contains}\ c\, \ulcorner v \urcorner \end{aligned}$$

A derivation of a judgment is represented by a canonical object of the corresponding type under the context obtained by translating the state. So if $\mathcal{E}x :: (S \triangleright K \vdash e \hookrightarrow w)$ then

$$\ulcorner S \urcorner \vdash_\Sigma \ulcorner \mathcal{E}x \urcorner \Uparrow \mathtt{exec}\, \ulcorner K \urcorner\, (\mathtt{ev}\, \ulcorner e \urcorner)\, \ulcorner w \urcorner$$

Conversely, a canonical object of such a type in a context of this form always represents a derivation of the corresponding judgment. In fact, the representation function $\ulcorner\ \urcorner$ is a compositional bijection between canonical *LLF* objects of the type above and derivations of the evaluation judgment. In this way *LLF* objects directly represent imperative computations.

As an example, we show the concrete representations of the rules that create cells (`ex_ref1`), derefence cells (`ex_deref1`), and change their value (`ex_assign2`). We use a single auxiliary judgment `read` with exactly one rule to obtain the contents of a cell without changing the state.

```
exec     : cont -> inst -> final -> type.
contains : cell -> val -> type.
read     : cell -> val -> type.

rd : read C V o- contains C V o- <T>.

ex_ref1 : exec K (ref1 V1) (new* W)
           o- ({c:cell} contains c V1
                -o exec K (return (ref* c)) (W c)).

ex_deref1 : exec K (deref1 (ref* C)) W
             o- read C V1 & exec K (return V1) W.

ex_assign2 : exec K (assign2 (ref* C1) V2) W
              o- contains C1 V1
              o- (contains C1 V2
                   -o exec K (return unit*) W).
```

The complete signature may be executed as a logic program with queries of the form `Ex : exec init (ev` $\ulcorner e \urcorner$`) W` for a closed expression e. If successful, `W` will be instantiated to the final answer and `Ex` to its computation; if it fails (which may happen if e is not well-typed in ML^{ref}) or does not terminate, then e has no value in the given operational semantics.

The last inference rule in the evaluation judgment in Figure 5 pairs up the state S with a value v to obtain the final answer (S, v). Since the state is represented in a distributed fashion, this must be implemented by a new type family, `close`, which transfers each cell from the context to the final answer.

The type preservation theorem relates the type of an expression to be evaluated to the type of the final answer. It is stated as follows:

Theorem (Type Preservation) *If* $S \triangleright K \vdash e \hookrightarrow w$ *and* $\Delta \vdash S : \Delta$, $\Delta \vdash K : \tau \Rightarrow \sigma$, *and* $\Delta \vdash e : \tau$ *then* $\Delta \vdash w : \sigma$.

Proof: The proof proceeds by structural induction on the derivation $\mathcal{E}x :: (S \triangleright K \vdash E \hookrightarrow w)$, applying inversion to obtain the typing derivations necessary to appeal to the induction hypothesis. □

While our development differs in some aspects from the formulations and proofs given in the literature (see, for example, [21] or [23]), our main contribution is not the proof itself, but its high-level implementation. We can indeed capture its computational contents but, as usual in *LF*, the fact that our code represents a proof needs to be checked as an external property [20]. The encoding takes the form of a relation between the derivations involved in the statement of type preservation. No lemmas or auxiliary judgments regarding state are required (except in the proof of the adequacy theorem for the representation) since the linear framework provides the necessary mechanisms internally. Space does not permit a detailed discussion, so we only show the declaration of the type family and the most complicated case. Each declaration in the full signature corresponds to exactly one case in the induction proof. The case given below treats the updating of a reference cell.

```
tpex : exec K I W -> ofk K T S
       -> ofi I T -> off W S -> type.

tpex_assign2 :
 tpex (ex_assign2 ^ ([et2^contains C1 V2] Ex2^et2)
       ^ Et1) Ok (ofi_assign2 Ov2 (ofv_ref Oc)) Of
    o- tpct Et1 Oc Ov1
    o- ({et2:contains C1 V2}
         tpct et2 Oc Ov2
         -o tpex (Ex2 ^ et2) Ok
                (ofi_return (ofv_unit)) Of).
```

Note that in this representation the first index object of `tpex` (*i.e.*, a derivation of `exec K I W`) is *linear*. In other words, we need the full power of the linear logical framework for this representation. Note that the meta-reasoning is also linear which stands in contrast to the cut elimination example given below, where the index objects (linear sequent derivations) are linear, but the meta-theory is implemented completely intuitionistically.

In the example above, the variable `et2` bound by the dependent type `{et2:contains C1 V2}` is quantified intuitionistically because the framework lacks a linear quantifier. The application `(Ex2 ^ et2)` in the first argument of `tpex` is nonetheless well-typed, since we can apply a linear function (`Ex2`) to intuitionistic arguments (`et2`), but not *vice versa*. For closed computations (and these are the ones we are ultimately interested in) this is irrelevant and thus the intuitionistic quantifier suffices for this and many other examples we have examined.

5 Another Example: Cut Elimination in Classical Linear Logic

A structural proof of the cut-elimination theorem for classical linear logic was given in [18], together with an *LF* formulation of a sequent calculus for it. Since *LF* is intuitionistic, the first step of this encoding consisted in representing the inference rules of this logic without concern for the restrictions they impose on the use of context formulas. Only in a second phase were the resulting derivations checked for linearity. The complications originating from this indirection prevented a direct formalization of linear cut-elimination: the encoding of this procedure applied to generic derivations. Fortunately, the specific proof under consideration had the property that each case in it maintained linearity. Therefore, given a linear derivation $\mathcal{D}$, the represented procedure yielded an equivalent (and linear) cut-free derivation $\mathcal{D}'$. The linearity of $\mathcal{D}$ had however to be checked explicitly.

In this section, we encode the same problem in *LLF*. The novel features of our framework permit a faithful representation of the linearity constraints each inference rule imposes on its context formulas. Therefore, only correct linear derivations can be built, so that we can dispense with encoding the tedious linearity check. The cut-elimination procedure is implemented as in [18], with the difference that it operates on the representation of linear derivations as linear *LLF* objects, and that its correctness does not depend on side-conditions. The complete code for this example can be found in [3].

Axioms

$$\frac{}{\Psi; A \longrightarrow A; \Theta}\,\mathrm{I}$$

Connectives

$\mathbf{1}$

$$\frac{}{\Psi; \cdot \longrightarrow \mathbf{1}; \Theta}\,\mathbf{1}\mathrm{r} \qquad \frac{\Psi; \Gamma \longrightarrow \Delta; \Theta}{\Psi; \Gamma, \mathbf{1} \longrightarrow \Delta; \Theta}\,\mathbf{1}\mathrm{l}$$

$\otimes$

$$\frac{\Psi; \Gamma_1 \longrightarrow A, \Delta_1; \Theta \quad \Psi; \Gamma_2 \longrightarrow B, \Delta_2; \Theta}{\Psi; \Gamma_1, \Gamma_2 \longrightarrow A \otimes B, \Delta_1, \Delta_2; \Theta}\,\otimes\mathrm{r} \qquad \frac{\Psi; \Gamma, A, B \longrightarrow \Delta; \Theta}{\Psi; \Gamma, A \otimes B \longrightarrow \Delta; \Theta}\,\otimes\mathrm{l}$$

$\top$

$$\frac{}{\Psi; \Gamma \longrightarrow \top, \Delta; \Theta}\,\top\mathrm{r} \qquad (\text{No } \top\,\mathbf{l})$$

$\&$

$$\frac{\Psi; \Gamma \longrightarrow A, \Delta; \Theta \quad \Psi; \Gamma \longrightarrow B, \Delta; \Theta}{\Psi; \Gamma \longrightarrow A \,\&\, B, \Delta; \Theta}\,\&\mathrm{r} \qquad \frac{\Psi; \Gamma, A \longrightarrow \Delta; \Theta}{\Psi; \Gamma, A \,\&\, B \longrightarrow \Delta; \Theta}\,\&\mathrm{l}_1 \qquad \frac{\Psi; \Gamma, B \longrightarrow \Delta; \Theta}{\Psi; \Gamma, A \,\&\, B \longrightarrow \Delta; \Theta}\,\&\mathrm{l}_2$$

$\perp$

$$\frac{\Psi; \Gamma, A \longrightarrow \Delta; \Theta}{\Psi; \Gamma \longrightarrow A^{\perp}, \Delta; \Theta}\,{\perp}\mathrm{r} \qquad \frac{\Psi; \Gamma \longrightarrow A, \Delta; \Theta}{\Psi; \Gamma, A^{\perp} \longrightarrow \Delta; \Theta}\,{\perp}\mathrm{l}$$

$!$

$$\frac{\Psi; \cdot \longrightarrow A; \Theta}{\Psi; \cdot \longrightarrow !A; \Theta}\,!\mathrm{r} \qquad \frac{\Psi, A; \Gamma \longrightarrow \Delta; \Theta}{\Psi; \Gamma, !A \longrightarrow \Delta; \Theta}\,!\mathrm{l}$$

$?$

$$\frac{\Psi; \Gamma \longrightarrow \Delta; A, \Theta}{\Psi; \Gamma \longrightarrow ?A, \Delta; \Theta}\,?\mathrm{r} \qquad \frac{\Psi; A \longrightarrow \cdot; \Theta}{\Psi; ?A \longrightarrow \cdot; \Theta}\,?\mathrm{l}$$

Structural rules

$$\frac{\Psi, A; \Gamma, A \longrightarrow \Delta; \Theta}{\Psi, A; \Gamma \longrightarrow \Delta; \Theta}\,!\mathrm{d} \qquad \frac{\Psi; \Gamma \longrightarrow A, \Delta; A, \Theta}{\Psi; \Gamma \longrightarrow \Delta; A, \Theta}\,?\mathrm{d}$$

Cut rules

$$\frac{\Psi; \Gamma_1 \longrightarrow A, \Delta_1; \Theta \quad \Psi; \Gamma_2, A \longrightarrow \Delta_2; \Theta}{\Psi; \Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2; \Theta}\,\mathrm{cut}$$

$$\frac{\Psi; \cdot \longrightarrow A; \Theta \quad \Psi, A; \Gamma \longrightarrow \Delta; \Theta}{\Psi; \Gamma \longrightarrow \Delta; \Theta}\,\mathrm{cut!} \qquad \frac{\Psi; \Gamma \longrightarrow \Delta; A, \Theta \quad \Psi; A \longrightarrow \cdot; \Theta}{\Psi; \Gamma \longrightarrow \Delta; \Theta}\,\mathrm{cut?}$$

Figure 6. The Sequent Calculus System LV

We consider the fragment of propositional classical linear logic defined by the following grammar:

$$A \;::=\; P \mid \mathbf{1} \mid A_1 \otimes A_2 \mid \top \mid A_1 \,\&\, A_2 \mid A^{\perp} \mid !A \;\mid ?A$$

where P ranges over propositional letters. The remaining connectives are all definable. A direct treatment would increase the length of proofs and encodings, but not their difficulty. A straightforward extension to encompass quantifiers adapts the techniques presented in [19] and is included in [3]. We encode formulas as canonical *LLF* objects of type `o`. As in the previous example, the linear aspects of our meta-language are not needed at this level.

As in [18, 19], we rely on the four-zoned sequents

$$\Psi; \Gamma \longrightarrow \Delta; \Theta$$

to capture the notion of derivability. Here, Ψ, Γ, Δ and Θ are multisets of implicitly labeled formulas. A deductive system for this formulation, that we call *LV*, is given in Figure 6. The *LV* sequent above is equiprovable to the more traditional $!\Psi, \Gamma \longrightarrow \Delta, ?\Theta$, as shown in [18]. This presentation isolates the intuitionistic reasoning in the outer zones of the sequents rather than restricting the structural rules of contraction and weakening to exponential formulas only. The removal of these rules is central to our proof of cut elimination. Notice that the distinction between linear and intuitionistic zones in *LV* causes the refinement of the familiar cut rule into three rules. As we will see, all are admissible.

We represent the occurrence of a formula in each zone of the sequent above by means of the following type families, respectively:

```
neg! : o -> type.      neg  : o -> type.
pos  : o -> type.      pos? : o -> type.
```

Derivability itself is captured by the type `#` with the intent that if $\mathcal{D}$ is a cut-free derivation of the sequent

$A_1, \ldots, A_i; B_1, \ldots, B_j \quad \longrightarrow \quad C_1, \ldots, C_k; D_1, \ldots, D_l$, with propositional letters among $p_1, \ldots, p_n$, then

$$\begin{bmatrix} p_1 : \texttt{o}, & \ldots, & p_n : \texttt{o}, \\ a_1 : \texttt{neg!}\ \ulcorner A_1 \urcorner, & \ldots, & a_i : \texttt{neg!}\ \ulcorner A_i \urcorner, \\ b_1 \hat{:} \texttt{neg}\ \ulcorner B_1 \urcorner, & \ldots, & b_j \hat{:} \texttt{neg}\ \ulcorner B_j \urcorner, \\ c_1 \hat{:} \texttt{pos}\ \ulcorner C_1 \urcorner, & \ldots, & c_k \hat{:} \texttt{pos}\ \ulcorner C_k \urcorner, \\ d_1 : \texttt{pos?}\ \ulcorner D_1 \urcorner, & \ldots, & d_l : \texttt{pos?}\ \ulcorner D_l \urcorner \end{bmatrix} \vdash_\Sigma \ulcorner \mathcal{D} \urcorner \Uparrow \texttt{\#}$$

is derivable in *LLF*. Notice that the formulas in the linear zones are represented as linear *LLF* assumptions while the formulas in the outer zones correspond to intuitionistic hypotheses. In order to achieve the representation above for derivations, we encode each inference rule so that it respects the zones policy. We show the *LLF* declarations corresponding to rules $\otimes\mathbf{r}$, $?\mathbf{l}$ and $!\mathbf{d}$:

```
timesr : (pos A -o #) -o (pos B -o #)
      -o (pos (A times B) -o #).
```

```
?l : (neg A -o #)
  -> (neg (? A) -o #).
```

```
!d : (neg A -o #)
  -o (neg! A -> #).
```

Notice that $\multimap$ at the level of derivations (i.e. just before `#`) implements access to the linear part of the context. In other positions, it permits the flow of linear assumptions among the premisses and conclusion of the represented rule. Instead, $\rightarrow$ allows accessing intuitionistic hypotheses and enforces constraints on the contents of the linear context (e.g. in `!d`).

The critical step in proving cut elimination is showing that the cut rules are admissible, i.e. that any (cut-free) derivation of their premisses can be transformed into a cut-free derivation of their conclusion. Formally, we have the following theorem:

Theorem (Admissibility of `cut`, `cut!` and `cut?`)

i. *Given cut-free derivations* $\mathcal{D} :: \Psi; \Gamma_1 \longrightarrow A, \Delta_1; \Theta$ *and* $\mathcal{E} :: \Psi; \Gamma_2, A \longrightarrow \Delta_2; \Theta$, *there exists a cut-free derivation* $\mathcal{F}$ *of* $\Psi; \Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2; \Theta$;

ii. *Given cut-free derivations* $\mathcal{D} :: \Psi; \cdot \longrightarrow A; \Theta$ *and* $\mathcal{E} :: \Psi, A; \Gamma \longrightarrow \Delta; \Theta$, *there exists a cut-free derivation* $\mathcal{F}$ *of* $\Psi; \Gamma \longrightarrow \Delta; \Theta$;

iii. *Given cut-free derivations* $\mathcal{D} :: \Psi; \Gamma \longrightarrow \Delta; A, \Theta$ *and* $\mathcal{E} :: \Psi; A \longrightarrow \cdot; \Theta$, *there exists a cut-free derivation* $\mathcal{F}$ *of* $\Psi; \Gamma \longrightarrow \Delta; \Theta$.

Proof: The three parts of this statement are proved simultaneously by induction on the lexicographic order defined by 1) the structure of A, 2) the convention that (*i*) is smaller than both (*ii*) and (*iii*), 3) the structure of $\mathcal{D}$ and $\mathcal{E}$ in the case of (*i*), of only $\mathcal{E}$ for (*ii*), and of only $\mathcal{D}$ in the case of (*iii*). In order to make this induction truly structural we have to consider linear proof terms instead of full derivations $\mathcal{D}$ and $\mathcal{E}$; further technical details are given in [18]. □

We transliterate the statement of the admissibility lemma as the type families

```
ad : (pos A  -o #) -> (neg A  -o #) -> # -> type.
ad!: (pos A  -o #) -> (neg! A -> #) -> # -> type.
ad?: (pos? A -> #) -> (neg A  -o #) -> # -> type.
```

Each case in the proof above is represented by an *LLF* declaration of the appropriate type. We show three declarations:

```
ad_times :
  ad ([p^]timesr ^ ([p1^]D1^p1) ^ ([p2^]D2^p2) ^p)
     ([n^]timesl ^ ([n1^]E1^n1^n)) F
   <- ({n2}ad ([p1^]D1^p1) ([n1^]E1^n1^n2) (E1'^n2))
   <- ad ([p2^]D2^p2) ([n1^]E1'^n1) F.
```

```
ad!_d :
  ad! ([p^]D!^p) ([n!] !d ^ ([n^]E1 n!^n) n!) F
   <- ({n1}ad! ([p^]D!^p) ([n!]E1 n!^n1) (E1'^n1))
   <- ad ([p^]D!^p) ([n1^]E1'^n1) F.
```

```
adl_withr :
  ad ([p^]withr^([p1^]D1^p^p1,[p2^]D2^p^p2)^P)
     ([n^]E^n) (withr^([p1^]D1'^p1,[p2^]D2'^p2)^P)
   <- ({p1} ad ([p^]D1^p^p1) ([n^]E^n) (D1' ^ p1))
    & ({p2} ad ([p^]D2^p^p2) ([n^]E^n) (D2' ^ p2))
```

As usual, the cut elimination theorem follows by a simple structural induction from admissibility. It can also be formalized in *LLF* as shown in [3]; we omit the straightforward details here.

6 Conclusion and Future Work

In this paper, we have developed an *LF*-style type theory based on linear logic and provided examples of its use as a logical framework in the theories of programming languages and substructural logics. Additional substantial case studies we have completed include translations between minimal linear natural deduction and sequent calculus, as well as a number of puzzles and solitaires. The interested reader may access them on the World-Wide Web at [3].

LLF conservatively extends *LF* with constructs from linear logic. We can think of it as the type theory freely generated from the type constructors $\top$, &, $\multimap$, and Π. This choice of constructors is complete in the sense that they suffice to represent full intuitionistic or classical linear logic, as our example in Section 5 shows. Further, adding any other linear connective as a free type constructor destroys the property that canonical forms exist. This property is crucial in the proofs of adequacy theorems for encodings and also for the completeness of uniform derivations and thus the view of *LLF* as a logic programming language.

LLF generalizes other formalisms based on linear logic such as *Forum* [12] by making linear objects available for representations, by permitting proof terms and by providing linear types. Our approach is orthogonal to general logics in the style of *LU* [8].

A slightly unpleasant feature of the type theory in its current form, from the practical point of view, is the insistence on dealing only with pre-canonical forms. Ideally the input would be η-expanded to long form, say, concurrently with type reconstruction and then the long form would be maintained throughout the computation as proposed here. We plan to consider such an extension in future work. Another possible extension we intend to investigate is a generalization of & and $\multimap$ to linear Σ and Π types, respectively. It currently appears that this would greatly complicate the type theory while it is not clear how much would be gained. We would also like to explore the possibility of automatically verifying that a signature implements a meta-proof of a meta-theorem analogously to *schema-checking* for *LF* [20].

In the more immediate future, we plan to release a concrete implementation of *LLF* as a conservative extension of the logic programming language *Elf* (which mechanizes *LF*) and to augment our already rich library of *LLF* examples.

References

[1] I. Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, Feb. 1996.

[2] I. Cervesato, J. S. Hodas, and F. 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, Mar. 1996. Springer-Verlag LNAI 1050.

[3] I. Cervesato and F. Pfenning. The linear logical framework *LLF*. Accessible on the World-Wide Web as `http://www.cs.cmu.edu/~iliano/LLF/`.

[4] J. L. Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[5] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, Sept. 1992.

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

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

[8] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[10] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[11] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, Jan. 1991. Springer-Verlag LNAI 596.

[12] D. Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994. IEEE Computer Society Press.

[13] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[14] D. Miller, G. Plotkin, and D. Pym. A relevant analysis of natural deduction. Talk given at the Workshop on Logical Frameworks, Båstad, Sweden, May 1992.

[15] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[16] F. Pfenning. Computation and deduction. Unpublished lecture notes, revised May 1994, May 1992.

[17] F. 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.

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

[19] F. Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[20] E. Rohwedder. *Verifying the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 1996. Forthcoming.

[21] M. Tofte. Type inference for polymorphic references. *Information & Computation*, 89:1–34, November 1990.

[22] P. Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, Apr. 1990. North-Holland.

[23] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information & Computation*, 115(1):38–94, November 1994.