The Linear Logical Framework

LLF

Iliano Cervesato

Department of Computer Science
Stanford University

(joint work with Frank Pfenning)
Contents

- Overview
- Logical frameworks
- LLF
- Case study
- Implementation
- Future Work
Overview

A **Logical Framework** is a formalism designed to represent and reason about deductive systems

**Aim:**

- identify the principles underlying logics and programming languages
  
  [Harper,Honsell,Plotkin’87; Pfenning’92; Michaylov,Pfenning’91; Shankar’94; Pfenning’95]

**Intended applications:**

- design of new and better logics and programming languages
- program verification and certification [Necula’97; Paulson’96]

**Limitations:**

- ineffective with imperative formalisms [Pfenning’94]
State

Till 2 years ago, **no** simple, general and effective treatment of the recurring notion of **state**

- store of an imperative programming language
- database
- communication among concurrent processes, ...

... **Linear Logic** [Girard’87]

- adequate for **representing** state and imperative computation [Chirimar’95; Hodas, Miller’94; Wadler’90]
- ineffective for **reasoning** about them
Achievements

- Design of a formalism, $LLF$, that combines
  - the meta-reasoning power of traditional logical frameworks
  - the possibility of linear logic of handling state

- Based on a *linear type theory*

- Conservative over $LF$ [Harper,Honsell,Plotkin’93]

- Used to represent
  - imperative programming languages
  - substructural logics
  - games, ...

  and to *reason* about them
Logical Frameworks

Formalisms specially designed to provide effective meta-representations of formal systems

**formal system**

programming languages, logics, ...

**meta-representation**

represent language constructs, model their semantics, encode properties and their proofs

**effectiveness**

immediacy and executability

| Logical framework = meta-language + representation methodology |
An Example: LF (Meta-Language)

• Syntax

\[ Kinds \quad K ::= \text{type} \mid \Pi x : A. K \]

\[ Type \ families \quad P ::= a \mid PM \]

\[ Types \quad A ::= P \mid \Pi x : A. B \]

\[ Objects \quad M ::= x \mid c \mid \lambda x : A. M \mid MN \]

• Typing judgment

\[ \Gamma \vdash_{\Sigma} M : A \quad \text{"M has type A in } \Gamma \text{ and } \Sigma" \]
An Example: LF (Meta-Language—Cont’d)

\[
\begin{align*}
\Gamma, x : A & \vdash \Sigma M : B \\
\Gamma & \vdash \Sigma \lambda x : A. M : \Pi x : A. B \\
\Gamma & \vdash \Sigma M : \Pi x : A. B \\
\Gamma & \vdash \Sigma N : A \\
\Gamma & \vdash \Sigma M N : [N/x]B
\end{align*}
\]

- Main properties
  - is strongly normalizing
  - admits unique canonical forms
  - type checking is decidable
  - can be implemented as a logic programming language (*Elf* [Pfenning’94])
An Example: LF (Representation Methodology)

Judgments-as-Types / Derivations-as-Objects

- Each object judgment is represented as a base type
- The context of an object judgment is encoded in the context of the meta-language
- Object-level inference rules are represented as constants that map derivations of their premisses to a derivation of their conclusion
- Derivations of an object judgment are represented as canonical terms of the corresponding base type
An Example: \textit{LF} (Representation Methodology—Cont’d)

\[
\begin{align*}
\Gamma & \vdash \tau \\
\Omega & \vdash e : \tau = M
\end{align*}
\]

\[
\Gamma \Omega \vdash \Sigma M : \text{has\_type} \Gamma e \Gamma \tau
\]

where for each \( x_i: \tau_i \) in \( \Omega \),

\[
\Gamma x_i : \tau_i = x_i : \text{exp}, \ t_i : \text{has\_type} x_i \Gamma \tau_i
\]

- context operations reduce to meta-level primitives
- meta-theoretic properties are inherited from the meta-language
Problem!

\[
\begin{array}{c}
c_i = v_i, \ldots \Rightarrow \\
\varepsilon \\
S \triangleright K \vdash e \leftrightarrow a = M
\end{array}
\]

\[
\langle S \rangle \vdash_\Sigma M : \text{eval} \langle K \rangle \langle e \rangle \langle a \rangle
\]

This does not work!

- $S$ is subject to destructive operations (e.g. assignment)
- traditional log. frameworks do not allow removing assumptions from the context

A way out ...

\[
\cdot \vdash_\Sigma M : \text{eval} \langle S \rangle \langle K \rangle \langle e \rangle \langle a \rangle
\]

... but, we must encode explicitly

- context operations (lookup, insertion, ...)
- context-related properties (weakening, exchange, ...)

Ilario Cervesato — The Linear Logical Framework LLF
LLF

- **Meta-language**: $\lambda^{\Pi -\&\top}$, a type theory based on $\Pi$, $\neg$, & and $\top$

- **Representation methodology**: judgments-as-types, but provides direct encoding of state in the linear context

- **Range of applicability**: declarative and imperative formalisms
**Syntax**

- **Kinds**
  \[ K ::= \text{type} \mid \Pi x : A. K \]

- **Type families**
  \[ P ::= a \mid P \ M \]

- **Types**
  \[ A ::= P \mid \Pi x : A. B \]
  \[ A \to B \mid A \& B \mid \top \]

- **Objects**
  \[ M ::= x \mid c \mid \lambda x : A. M \mid M \ N \]
  \[ \hat{\lambda} x : A. M \mid M \ ^\top \ N \mid \langle M, N \rangle \mid \text{fst} \ M \mid \text{snd} \ M \mid \langle \rangle \]

**Typing judgment**

\[ \Gamma; \Delta \vdash \Sigma \ M : A \]

"\( M \) has type \( A \) in \( \Gamma, \Delta \) and \( \Sigma \)"

**Linear context**

\[ x : A, \ldots \]

**Intuitionistic context**

\[ x : A, \ldots \]

**Signature**

\[ a : K, \ldots, c : A, \ldots \]
Some Inference Rules

\[ \chi^\Pi_{\circ \& \top}, \text{Some Inference Rules} \]

\[
\frac{\Gamma, x : A; \cdot \vdash \Sigma x : A}{\text{ivar}} \quad \frac{\Gamma, x \uparrow A \vdash \Sigma x : A}{\text{ivar}}
\]

\[
\frac{\Gamma, x : A; \Delta \vdash \Sigma M : B}{\Gamma; \Delta \vdash \Sigma \lambda x : A. M : \Pi x : A. B} \quad \frac{\Gamma; \Delta \vdash \Sigma M N : [N/x]B}{\text{iapp}}
\]

\[
\frac{\Gamma; \Delta, \uparrow x A \vdash \Sigma M : B}{\Gamma; \Delta \vdash \Sigma \hat{x} : A. M : A \rightarrow B} \quad \frac{\Gamma; \Delta_1 \vdash \Sigma M : A \rightarrow B}{\Gamma; \Delta_1, \Delta_2 \vdash \Sigma M N : B} \quad \frac{\Gamma; \Delta \vdash \Sigma N : A}{\text{iapp}}
\]

\[
\frac{\Gamma; \Delta \vdash \Sigma M : A}{\Gamma; \Delta \vdash \Sigma \langle M, N \rangle : A \& B} \quad \frac{\Gamma; \Delta \vdash \Sigma M : A \& B}{\text{pair}} \quad \frac{\Gamma; \Delta \vdash \Sigma \text{FST } M : A}{\text{fst}} \quad \frac{\Gamma; \Delta \vdash \Sigma \text{SND } M : B}{\text{snd}}
\]

\[
\frac{\Gamma; \Delta \vdash \Sigma \langle \rangle : \top}{\text{unit}}
\]
$LLF$, Main Properties

- Church-Rosser property
- strongly normalizing
- unique canonical forms
- decidability of type checking
- abstract logic programming language
- conservative over $LF$
Immediacy in $LLF$

Direct correlation between an object system and its encoding.

$LLF$ gives direct support to recurrent representation patterns:

- binding constructs via $\lambda$-abstraction
- derivations as proof-terms
- state manipulation via linear constructs
Case Study: **MLR**

**MLR** is a fragment of **ML** with

- references
- value polymorphism
- recursion

$$
\text{Types } \tau ::= \ldots \mid 1 \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ ref}
$$

$$
\text{Expressions } e ::= x
\mid \langle \rangle
\mid \text{lam } x.e
\mid e_1 \ e_2
\mid \ldots
\mid c
\mid \text{ref } e
\mid !e
\mid e_1 := e_2
$$

$$
\text{Store } S ::= \cdot \mid S, c = v
$$

<table>
<thead>
<tr>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp : type.</td>
</tr>
<tr>
<td>cell : type.</td>
</tr>
<tr>
<td>unit : exp.</td>
</tr>
<tr>
<td>lam : (exp -&gt; exp) -&gt; exp.</td>
</tr>
<tr>
<td>app : exp -&gt; exp -&gt; exp.</td>
</tr>
<tr>
<td>loc : cell -&gt; exp.</td>
</tr>
<tr>
<td>ref : exp -&gt; exp.</td>
</tr>
<tr>
<td>deref : exp -&gt; exp.</td>
</tr>
<tr>
<td>assign : exp -&gt; exp -&gt; exp.</td>
</tr>
</tbody>
</table>
**MLR: Typing**

\[ \Omega \vdash e : \tau \quad \text{"e has type } \tau \text{ in } \Omega \text{"} \]

**Context** \( x_i : \tau_i, \ldots, c_j : \sigma_j, \ldots \)

**Expression** \( \text{Expression} \)

**Type** \( \text{Type} \)

**Representation:**

\[ \Gamma \vdash \Sigma \quad \Gamma e : \text{exp_type} \Gamma e \Gamma \tau \]

\( x_i : \text{exp}, \ t_i : \text{exp_type} x_i \Gamma \tau_i, \ldots \)

\( c_j : \text{cell}, \ l_j : \text{cell_type} c_j \Gamma \sigma_j, \ldots \)

**Inference Rules:**

- **et_assign**

\[ \Omega \vdash e_1 : \tau \text{ ref} \quad \Omega \vdash e_2 : \tau \]

\[ \Omega \vdash e_1 := e_2 : 1 \]

- **et_deref**

\[ \Omega \vdash e : \tau \text{ ref} \]

\[ \Omega \vdash !e : \tau \]

- **et_assign**

\[ \text{et_assign} : \text{exp_type } E1 \text{ (rf T)} \rightarrow \text{exp_type } E2 \text{ T} \rightarrow \text{exp_type } \text{assign } E1 \text{ E2} \text{ 1}. \]

- **et_deref**

\[ \text{et_deref} : \text{exp_type } E \text{ (rf T)} \rightarrow \text{exp_type } \text{deref } E \text{ T}. \]
**MLR: Evaluation**

\[
S \triangleright K \vdash i \rightarrow a
\]

"\(i\) followed by \(K\) evaluates to \(a\), starting from \(S\)"

**Continuation**

\[
\text{init}, \ldots, \lambda x. i, \ldots
\]

**Instruction**

\[
\text{eval } e, \quad \text{return } v, \ldots
\]

**Store**

\[
c_i = v_i, \ldots
\]

**Answer**

Representation:

\[
\Gamma \vdash S \vdash \Sigma \vdash \mathcal{E} \vdash \text{eval} \vdash K \vdash i \vdash a
\]

\[
c_i : \text{cell}, \quad h_i \uparrow \text{contains} \ c_i \vdash v_i, \ldots
\]
**MLR: Some Imperative Rules**

$$S', c = v, S'' \triangleright K \vdash \text{return } \langle \rangle \mapsto a$$

$$S', c = v', S'' \triangleright K \vdash c := v \mapsto a$$

**ev_assign**

- (contains C V) -o eval K (return unit) A
- -o (contains C V') -o eval K (assign2 (loc C) V) A

$$S', c = v, S'' \triangleright K \vdash \text{return } v \mapsto a$$

$$S', c = v, S'' \triangleright K \vdash !c \mapsto a$$

**ev_deref**

- read C V
- & eval K (return V) A
- -o eval K (ref1 (loc C)) A

**rd**

- contains C V
- -o <T>
- -o read C V.
**MLR: Adequacy**

**Adequacy theorem (Evaluation)**

Given a store $S = (c_1 = v_1, \ldots, c_n = v_n)$, a continuation $K$, an instruction $i$ and an answer $a$, all closed, there is a bijection between derivations $\mathcal{E}$ of

$$S \triangleright K \vdash i \leftrightarrow a$$

and canonical $LLF$ objects $M$ such that

$$\Gamma S \vdash \Sigma M : \text{eval} \Gamma K \Gamma i \Gamma a$$

is derivable, where

$$\Gamma S = \begin{bmatrix}
  c_1 : \text{cell}, & h_1 \uparrow \text{contains} & c_1 \Gamma v_1 \\
  \cdots \\
  c_n : \text{cell}, & h_n \uparrow \text{contains} & c_n \Gamma v_n
\end{bmatrix}$$
**MLR: Type Preservation**

- Functional core: implemented in $LF$ [Michaylov,Pfenning’91]
- References [Tofte’90; Harper’94]: implemented in $LLF$ [Cervesato’96]

**Theorem** (*type preservation*)

If $S \triangleright K \vdash i \leftrightarrow a$, with $\Omega \vdash i : \tau$, $\Omega \vdash K : \tau \Rightarrow \sigma$ and $\Omega \vdash S : \Omega$, then $\Omega \vdash a : \sigma$

**Proof**: by induction on the evaluation derivation

The high level of abstraction of the representation permits *transcribing* this proof into an $LLF$ specification capturing its computational contents

- each case yields one declaration
- the meta-reasoning is itself *linear*

**Representation**

\[ \text{tpev} : \text{eval} \ K \ I \ A \rightarrow \text{cont}_\text{type} \ K \ T \ S \rightarrow \text{instr}_\text{type} \ I \ T \rightarrow \text{ans}_\text{type} \ A \ S \rightarrow \text{type}. \]
Implementation

---

**LLF** is implemented as part of the *Twelf* project

- **Twelf, a successor to *Elf*** [Pfenning’94]
  - higher-order constraint logic programming language based on *LF* and *LLF*
  - automated theorem prover in a meta-logic for *LF* [Schürmann,Pfenning’98]
  - internals: explicit substitutions, spine calculus, compilation

- **Linear aspects**
  - linearity check
  - resource management [Hodas,Miller’94; Cervesato,Pfenning’96]
  - linear unification [Cervesato,Pfenning’97]
**LLF, Summary**

- combines the meta-reasoning power of logical frameworks with the ability of handling state of linear logic
- conservative extension of the logical framework *LF*
- implemented as a linear logic programming language
- used for the representation of
  - imperative programming languages
  - substructural and modal logics
  - puzzles and solitaires
  - planning
  - imperative graph search
Future Work

• Specification and verification of
  – “real” programming languages (e.g. SML’97, Java)
  – communication protocols
  – logics

• Proof-Carrying Code [Necula’97]

• Computer-assisted development environments for logics and programming languages (meta-logical frameworks)

• Type theoretic extensions of LLF (e.g. dependent linear types [Ishtiaq,Pym’97], non-commutativity [Pfenning’98])