The Linear Logical Framework

\textit{LLF}

Iliano Cervesato

Department of Computer Science
Stanford University

(joint work with Frank Pfenning)
Contents

- Overview
- Logical frameworks
- LLF
- Case study
- Conclusions
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, \textit{LLF}, that combines
  - the meta-reasoning power of traditional logical frameworks
  - the possibility of linear logic of handling state
- Based on a \textit{linear type theory}
- Conservative over \textit{LF} [Harper,Honsell,Plotkin’93]
- Used to represent
  - imperative programming languages
  - substructural logics
  - games, ...
  and to \textit{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**

  \[
  \text{Kinds} \quad K ::\text{ type} \mid \Pi x : A. K \\
  \text{Type families} \quad P ::\text{ a} \mid P M \\
  \text{Types} \quad A :: P \mid \Pi x : A. B \\
  \text{Objects} \quad M :: x \mid c \mid \lambda x : A. M \mid M N
  \]

- **Typing judgment**

  \[
  \Gamma \vdash_{\Sigma} M : A \\
  \text{“}M \text{ has type } A \text{ in } \Gamma \text{ and } \Sigma \text{”}
  \]

  \[
  \begin{array}{c}
  \text{Context} \\
  x : A, \ldots
  \end{array} \quad \begin{array}{c}
  \text{Signature} \\
  a : K, \ldots, c : A, \ldots
  \end{array}
  \]
An Example: \( LF \) (Meta-Language—Cont’d)

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

- **Main properties**
  - is strongly normalizing
  - admits unique canonical forms
  - type checking is decidable
  - can be implemented as a logic programming language (\textit{Elf} \cite{Pfenning94})
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)

\[
\begin{align*}
\begin{array}{c}
\text{x_i:}\tau_i, \ldots \quad \mathcal{T} \\
\Omega \vdash e : \tau
\end{array}
\end{align*}
\]

\[
\Gamma \vdash \Sigma \ M : \text{has_type} \ \Gamma e \ \Gamma \tau
\]

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

\( \Gamma \vdash x_i : \tau_i = x_i : \text{exp}, \ t_i : \text{has_type} \ x_i \ \Gamma \tau \)

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

\[ c_i = v_i, \ldots \quad \vdash \quad \mathcal{E} \quad \Downarrow \quad S \triangleright K \vdash e \leftrightarrow a = M \]

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

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 ...

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

... but, we must encode explicitly

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

Ilario Cervesato — The Linear Logical Framework LLF
• **Meta-language**: $\lambda^{\Pi-\circ & \top}$, a type theory based on $\Pi$, $\circ$, $\&$ and $\top$

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

• **Range of applicability**: declarative and *imperative* formalisms
**Linear Logic in Brief**

\[ \Gamma; \Delta \vdash A \]

Main resource operators

- \( A \otimes B \) = “\( A \) and \( B \) simultaneously”
- \( A \& B \) = “\( A \) and \( B \) alternatively”
- \( T \) = “resource sink”
- \( A \circ B \) = “\( B \) assuming \( A \) as a resource”
- \( A \rightarrow B \) = “\( B \) assuming \( A \) as a logical hypothesis”

Accessing a resource consumes it
\( \lambda^\Pi \to \& \top \), the Meta-Language of LLF

- **Syntax**

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

  \[ P \ ::= a \mid P M \]

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

  \[ A \to B \mid A \& B \mid \top \]

  \[ M \ ::= x \mid c \mid \lambda x : A . M \mid M N \]

  \[ \widehat{\lambda} x : A . M \mid M^*N \mid \langle M, N \rangle \mid \text{FST } M \mid \text{SND } M \mid \emptyset \]

- **Typing judgment**

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

  "\( M \) has type \( A \) in \( \Gamma, \Delta \) and \( \Sigma \)"
$LLF$, 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: \textit{MLR}

\textit{MLR} is a fragment of \textit{ML} with

- references
- value polymorphism
- recursion

\textit{Types} \quad \tau ::= \ldots \mid 1 \mid \tau_1 \to \tau_2 \mid \tau \text{ ref}

\textit{Expressions} \quad e ::= x
\quad | \langle \rangle
\quad | \text{lam} \ x. \ e
\quad | e_1 \ e_2
\quad | \ldots
\quad | c
\quad | \text{ref} \ e
\quad | \! e
\quad | e_1 := e_2

\textit{Store} \quad S ::= \cdot \mid S, c = v

<table>
<thead>
<tr>
<th>\text{Expressions}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{exp} : \text{type}.</td>
</tr>
<tr>
<td>\text{cell} : \text{type}.</td>
</tr>
<tr>
<td>\text{unit} : \text{exp}.</td>
</tr>
<tr>
<td>\text{lam} : (\text{exp} \to \text{exp}) \to \text{exp}.</td>
</tr>
<tr>
<td>\text{app} : \text{exp} \to \text{exp} \to \text{exp}.</td>
</tr>
<tr>
<td>\ldots</td>
</tr>
<tr>
<td>\text{loc} : \text{cell} \to \text{exp}.</td>
</tr>
<tr>
<td>\text{ref} : \text{exp} \to \text{exp}.</td>
</tr>
<tr>
<td>\text{deref} : \text{exp} \to \text{exp}.</td>
</tr>
<tr>
<td>\text{assign} : \text{exp} \to \text{exp} \to \text{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**

\[ x_i : \text{exp}, \quad t_i : \text{exp}_\text{type} \quad x_i^{\tau_i}, \quad \ldots \]

\[ c_j : \text{cell}, \quad l_j : \text{cell}_\text{type} \quad c_j^{\sigma_j}, \quad \ldots \]

**Type**

\[ \Gamma \vdash \Sigma \quad \Gamma : \text{exp}_\text{type} \quad e^{\tau} \quad \tau \]

**Representation:**

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

\[ \Omega \vdash e_1 := e_2 : \mathbf{1} \quad \text{et\_assign} \]

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

\[ \Omega \vdash !e : \tau \quad \text{et\_deref} \]

**Rule Definitions:**

- **et\_assign:**
  \[ \begin{align*}
  \text{et\_assign} & : \text{exp}_\text{type} \ E1 \ (\text{rf T}) \\
  & \to \text{exp}_\text{type} \ E2 \ T \\
  & \to \text{exp}_\text{type} \ (\text{assign E1 E2}) \mathbf{1}.
  \end{align*} \]

- **et\_deref:**
  \[ \begin{align*}
  \text{et\_deref} & : \text{exp}_\text{type} \ E \ (\text{rf T}) \\
  & \to \text{exp}_\text{type} \ (\text{deref E}) \ T.
  \end{align*} \]
**MLR: Evaluation**

Continuation
\( \text{init}, \ldots, \lambda x. i, \ldots \)

Instruction
\( \text{eval } e, \text{ return } v, \ldots \)

\( S \triangleright K \vdash i \rightarrow a \)

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

Store
\( c_i = v_i, \ldots \)

Answer

Representation:

\[ \Gamma \mid S \vdash_{\Sigma} \text{eval } \Gamma K \mid i \mid a \]

\( c_i : \text{cell}, \ h_i \hat{\ : \ contains } \ c_i \hat{\mid } v_i \hat{\mid }, \ldots \)
**MLR: Some Imperative Rules**

\[
S', c = v, S'' \triangleright K \vdash \text{return} \langle \rangle \mapsto a \quad \text{ev_assign}
\]

\[
S', c = v', S'' \triangleright K \vdash c := v \mapsto a \quad \text{ev_assign}
\]

\[
\text{ev_assign :} \quad \begin{cases} 
(\text{contains C V} & \to \text{eval K (return unit) A}) \\
(\text{contains C V'} & \to \text{eval K (assign2 (loc C) V) A})
\end{cases}
\]

\[
S', c = v, S'' \triangleright K \vdash \text{return} v \mapsto a \quad \text{ev_deref}
\]

\[
S', c = v, S'' \triangleright K \vdash !c \mapsto a \quad \text{ev_deref}
\]

\[
\text{ev_deref :} \quad \begin{cases} 
\text{read C V} \\
& \& \text{eval K (return V) A} \\
& \to \text{eval K (ref1 (loc C)) A}
\end{cases}
\]

\[
\text{rd :} \quad \begin{cases} 
\text{contains C V} \\
& \to <T> \\
& \to \text{read C V}
\end{cases}
\]
**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

$$\Gamma S \Gamma = \begin{bmatrix}
  c_1 : \text{cell}, & h_1 \hat{\text{contains}} c_1 \Gamma v_1 \\
  \vdots \\
  c_n : \text{cell}, & h_n \hat{\text{contains}} c_n \Gamma v_n
\end{bmatrix}$$
**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**

```
tpev : eval K I A -> cont_type K T S -> instr_type I T -> ans_type A S -> type.
```
Implementation

\(LLF\) is implemented as part of the \textit{Twelf} project

- **Twelf, a successor to \textit{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]
\textbf{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 \textit{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])