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**

$$
\begin{array}{rrcl}
\textit{Kinds} & K & ::= & \mathtt{type} \mid \Pi x{:}A.\,K \\
\textit{Type families} & P & ::= & a \mid P\,M \\
\textit{Types} & A & ::= & P \mid \Pi x{:}A.\,B \\
\textit{Objects} & M & ::= & x \mid c \mid \lambda x{:}A.\,M \mid M\,N
\end{array}
$$

- **Typing judgment**

$$\Gamma \vdash_\Sigma M : A$$

"M has type A in Γ and Σ"

Context
$x{:}A, \ldots$

Signature
$a{:}K, \ldots, c{:}A, \ldots$

An Example: *LF* (Meta-Language—Cont'd)

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x{:}A.\, M : \Pi x{:}A.\, B}\,\textbf{lam} \qquad \frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.\, B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\, N : [N/x]B}\,\textbf{app}$$

- **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: *LF* (Representation Methodology—Cont'd)

$$\ulcorner \overset{x_i{:}\tau_i, \ldots}{\underset{\Omega \vdash e : \tau}{\mathcal{T}}} \urcorner = M$$

$$\ulcorner\Omega\urcorner \vdash_\Sigma M : \texttt{has\_type}\,\ulcorner e\urcorner\,\ulcorner\tau\urcorner$$

where for each $x_i{:}\tau_i$ in Ω,

$$\ulcorner x_i{:}\tau_i\urcorner = x_i{:}\texttt{exp},\ t_i{:}\texttt{has\_type}\,x_i\,\ulcorner\tau_i\urcorner$$

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

Problem!

$c_i = v_i, \ldots$

$$\ulcorner \frac{\mathcal{E}}{S \triangleright K \vdash e \hookrightarrow a} \urcorner = M$$

$$\ulcorner S \urcorner \vdash_\Sigma M : \texttt{eval} \ulcorner K \urcorner \ulcorner e \urcorner \ulcorner a \urcorner$$

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 ...

$$\cdot \vdash_\Sigma M : \texttt{eval} \ulcorner S \urcorner \ulcorner K \urcorner \ulcorner e \urcorner \ulcorner a \urcorner$$

... but, we must encode *explicitly*

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

- **Meta-language**: $\lambda^{\Pi\multimap\&\top}$, a type theory based on Π, $\multimap$, $\&$ and $\top$
- **Representation methodology**: judgments-as-types, but provides direct encoding of state in the linear context
- **Range of applicability**: declarative and imperative formalisms

$\lambda^{\Pi\multimap\&\top}$, the Meta-Language of *LLF*

- **Syntax**

$$
\begin{array}{rrcl}
\textit{Kinds} & K & ::= & \mathtt{type} \mid \Pi x\!:\!A.\,K \\
\textit{Type families} & P & ::= & a \mid P\,M \\
\textit{Types} & A & ::= & P \mid \Pi x\!:\!A.\,B \\
 & & & \mid A \multimap B \mid A \,\&\, B \mid \top \\
\textit{Objects} & M & ::= & x \mid c \mid \lambda x\!:\!A.\,M \mid M\,N \\
 & & & \mid \hat{\lambda} x\!:\!A.\,M \mid M\,\hat{}\,N \mid \langle M, N\rangle \mid \textsc{fst}\,M \mid \textsc{snd}\,M \mid \langle\rangle
\end{array}
$$

- **Typing judgment**

Linear context

$x \mathbin{\hat{:}} A, \ldots$

$$\Gamma; \Delta \vdash_\Sigma M : A$$

"M has type A in Γ, Δ and Σ"

Intuitionistic context

$x\!:\!A, \ldots$

Signature

$a\!:\!K, \ldots, c\!:\!A, \ldots$

$\lambda^{\Pi \multimap \& \top}$, Some Inference Rules

$$\frac{}{\Gamma, x{:}A; \cdot \vdash_\Sigma x : A}\ \textbf{ivar} \qquad \frac{}{\Gamma; x \hat{:} A \vdash_\Sigma x : A}\ \textbf{lvar}$$

$$\frac{\Gamma, x{:}A; \Delta \vdash_\Sigma M : B}{\Gamma; \Delta \vdash_\Sigma \lambda x{:}A.\, M : \Pi x{:}A.\, B}\ \textbf{ilam} \qquad \frac{\Gamma; \Delta \vdash_\Sigma M : \Pi x{:}A.\, B \quad \Gamma; \cdot \vdash_\Sigma N : A}{\Gamma; \Delta \vdash_\Sigma M\, N : [N/x]B}\ \textbf{iapp}$$

$$\frac{\Gamma; \Delta, x \hat{:} A \vdash_\Sigma M : B}{\Gamma; \Delta \vdash_\Sigma \hat{\lambda} x{:}A.\, M : A \multimap B}\ \textbf{llam} \qquad \frac{\Gamma; \Delta_1 \vdash_\Sigma M : A \multimap B \quad \Gamma; \Delta_2 \vdash_\Sigma N : A}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma M \hat{}\, N : B}\ \textbf{iapp}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma M : A \quad \Gamma; \Delta \vdash_\Sigma N : B}{\Gamma; \Delta \vdash_\Sigma \langle M, N \rangle : A \,\&\, B}\ \textbf{pair} \qquad \frac{\Gamma; \Delta \vdash_\Sigma M : A \,\&\, B}{\Gamma; \Delta \vdash_\Sigma \textsc{fst}\, M : A}\ \textbf{fst} \qquad \frac{\Gamma; \Delta \vdash_\Sigma M : A \,\&\, B}{\Gamma; \Delta \vdash_\Sigma \textsc{snd}\, M : B}\ \textbf{snd}$$

$$\frac{}{\Gamma; \Delta \vdash_\Sigma \langle\rangle : \top}\ \textbf{unit}$$

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 λ-abstraction
- derivations as proof-terms
- state manipulation via linear constructs

Case Study: *MLR*

MLR is a fragment of *ML* with

- references
- value polymorphism
- recursion

$$
\begin{array}{rrcl}
\textit{Types} & \tau & ::= & \ldots \mid \mathbf{1} \mid \tau_1 \to \tau_2 \mid \tau\ \mathbf{ref} \\[1em]
\textit{Expressions} & e & ::= & x \\
 & & \mid & \langle\rangle \\
 & & \mid & \mathbf{lam}\ x.\, e \\
 & & \mid & e_1\, e_2 \\
 & & \mid & \ldots \\
 & & \mid & c \\
 & & \mid & \mathbf{ref}\ e \\
 & & \mid & \mathbf{!}e \\
 & & \mid & e_1 := e_2 \\[1em]
\textit{Store} & S & ::= & \cdot \mid S, c = v
\end{array}
$$

Expressions

```
exp  : type.
cell : type.

unit   : exp.
lam    : (exp -> exp) -> exp.
app    : exp -> exp -> exp.
       ...
loc    : cell -> exp.
ref    : exp -> exp.
deref  : exp -> exp.
assign : exp -> exp -> exp.
```

MLR: Typing

$$\Omega \vdash e : \tau \qquad \text{“}e \text{ has type } \tau \text{ in } \Omega\text{”}$$

Context: $x_i\!:\!\tau_i, \ldots, c_j\!:\!\sigma_j, \ldots$

Expression

Type

Representation: $\ulcorner\Omega\urcorner \vdash_\Sigma \ulcorner\mathcal{T}\urcorner : \texttt{exp\_type}\ulcorner e\urcorner\ulcorner\tau\urcorner$

$$\begin{array}{llll} x_i\!:\!\texttt{exp}, & t_i\!:\!\texttt{exp\_type}\, x_i\, \ulcorner\tau_i\urcorner, & \ldots \\ c_j\!:\!\texttt{cell}, & l_j\!:\!\texttt{cell\_type}\, c_j\, \ulcorner\sigma_j\urcorner, & \ldots \end{array}$$

$$\frac{\Omega \vdash e_1 : \tau\ \textbf{ref} \quad \Omega \vdash e_2 : \tau}{\Omega \vdash e_1 := e_2 : \mathbf{1}}\ \textbf{et\_assign}$$

```
et_assign :  exp_type E1 (rf T)
          -> exp_type E2 T
          -> exp_type (assign E1 E2) 1.
```

$$\frac{\Omega \vdash e : \tau\ \textbf{ref}}{\Omega \vdash \mathbf{!}e : \tau}\ \textbf{et\_deref}$$

```
et_deref  :  exp_type E (rf T)
          -> exp_type (deref E) T.
```

MLR: Evaluation

Continuation
init, $\ldots, \lambda x.\, i, \ldots$

Instruction
eval e,
return $v, \ldots$

$$S \rhd K \vdash i \hookrightarrow a$$

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

Store
$c_i = v_i, \ldots$

Answer

Representation: $\ulcorner S \urcorner \vdash_\Sigma \ulcorner \mathcal{E} \urcorner : \texttt{eval}\ \ulcorner K \urcorner\, \ulcorner i \urcorner\, \ulcorner a \urcorner$

$c_i : \texttt{cell},\ h_i \hat{:} \texttt{contains}\ c_i\, \ulcorner v_i \urcorner,\ \ldots$

MLR: Some Imperative Rules

$$\frac{S', c = v, S'' \rhd K \vdash \textbf{return}\ \langle\rangle \hookrightarrow a}{S', c = v', S'' \rhd K \vdash c := v \hookrightarrow a}\ \textbf{ev\_assign}$$

```
ev_assign :     (contains C V   -o  eval K (return unit) A)
             -o (contains C V'  -o  eval K (assign2 (loc C) V) A).
```

$$\frac{S', c = v, S'' \rhd K \vdash \textbf{return}\ v \hookrightarrow a}{S', c = v, S'' \rhd K \vdash\ !c \hookrightarrow a}\ \textbf{ev\_deref}$$

```
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.
```

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

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

and canonical *LLF* objects M such that

$$\ulcorner S \urcorner \vdash_\Sigma M : \mathtt{eval}\ulcorner K \urcorner \ulcorner i \urcorner \ulcorner a \urcorner$$

is derivable, where

$$\ulcorner S \urcorner = \begin{bmatrix} c_1 : \mathtt{cell}, & h_1 \hat{:} \mathtt{contains}\, c_1 \ulcorner v_1 \urcorner \\ & \ldots \\ c_n : \mathtt{cell}, & h_n \hat{:} \mathtt{contains}\, c_n \ulcorner v_n \urcorner \end{bmatrix}$$

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 $S \triangleright K \vdash i \hookrightarrow a$, 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 *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])