A Linear Logical Framework

Presentazione della

Tesi di Dottorato

Iliano Cervesato

Dipartimento di Informatica
Università di Torino
Corso Svizzera, 185
10149 Torino - Italy

`iliano@di.unito.it`

Questa ricerca è stata sviluppata durante un'estesa visita al Department of Computer Science *presso la* Carnegie Mellon University

Milano, 6 Febbraio 1996

Overview

- Outline of the problem
  - Effective representation of formal systems
  - Examples of applications
  - State of the art
  - Contributions
- Background
  - Deductive systems
  - Meta-representation
  - The logical framework *LF*
  - Linear logic
- Technical development
  - The linear logical framework *LLF*
  - Properties
  - Logic programming in *LLF*
- Examples
  - *MLR*: *Mini-ML* with references
  - Cut-elimination for linear logic
  - Mahjongg
- Conclusions and future work

The problem

Provide

Effective meta-representations of formal systems

- Formal systems: almost everything of interest in logic and computer science:
  - logics
  - programming languages
  - micro-chips, ...
- Meta-representation: an encoding of aspects of
  - the syntax
  - the semantics
  - the meta-theory

  of a formal system
- Effectiveness:
  - amenability to mechanization
  - usability

Example of applications

Example of applications

- Program transformation
- Design of hardware components
- Verification of logical proofs
- Design of programming languages

State of the art

A *logical framework* is a language specially tailored to give effective representations of formal systems

<u>Logics</u>

- Horn clauses (*Prolog*)
- Hereditary Harrop formulas (λ*Prolog*)

<u>Type theories</u>

- *AUTOMATH* languages (*AUTOMATH*)
- Martin-Löf's type theories (*NuPrl*, *ALF*)
- *LF* (*Elf*)
- Calculi of Constructions (*Coq*, *LEGO*)

They have been used successfully to encode

- traditional logics and type theories
- pure functional and logic programming languages

They are ineffective for

- logics with complex operations on the context
- programming languages based on a mutable state

Exception

Forum

No commercial applicability

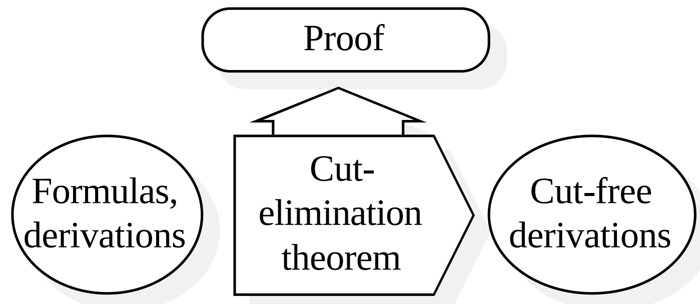
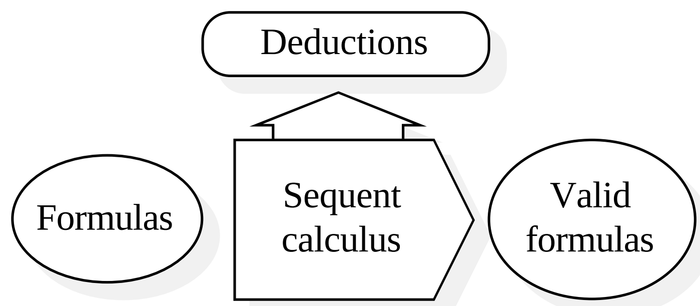
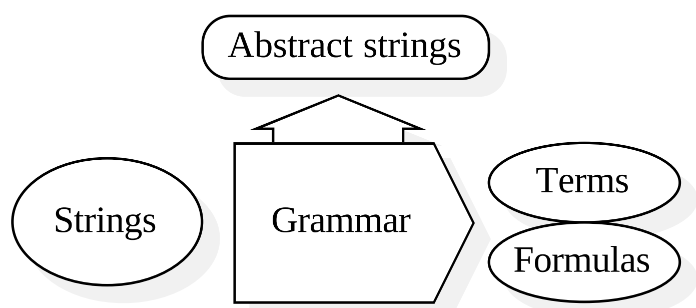
Contribution

The meta-representation formalism *LLF* is an extension of the logical framework *LF* with constructs from linear logic.

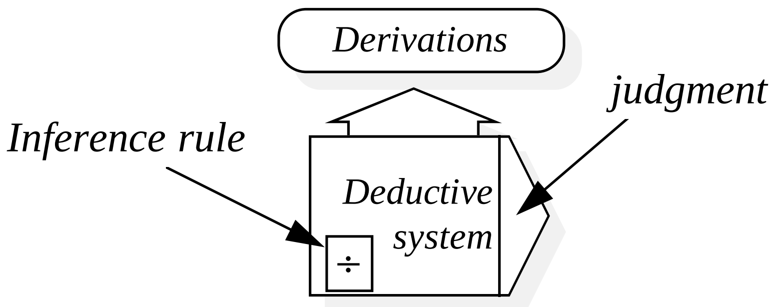
It permits representing and reasoning about:

- imperative programming languages characterized by a mutable store
- logics with a non-monotonic treatment of their context (e.g. linear logic)
- languages with linear binding constructs (*linear higher-order abstract syntax*)
- problems based on a state that evolves with time (e.g. puzzles and solitaires)

Structure of a formal system

...	...
<u>Meta-theory</u> *Properties* of valid objects from the semantic layer	
<u>Semantics</u> *Meaning* of the constructs in the language	
<u>Syntax</u> Membership of strings to the syntactic categories of the language	

Deductive systems



Judgment

Relation on lower level derivations

Context

"Abstract" formula

$$\Gamma \to A$$

Inference rule

Conditional derivability of judgments

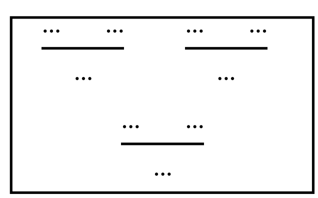
Premisses

$$\frac{\Gamma \to A \qquad \Gamma \to B}{\Gamma \to A \wedge B}\ \wedge R$$

Conclusion

Deductive system

Set of inference rules



Derivation

Evidence of the derivability of a judgment

$$\frac{\dfrac{}{A \to A}\,Ax \qquad \dfrac{}{A \to T}\,T\,R}{A \to A \wedge T}\ \wedge R$$

$$\boxed{\wedge R\ (Ax,\ T\,R)}$$

Anatomy of a deductive system

$\dfrac{}{\Gamma \to C}$	"C is derivable"	C
$\dfrac{\Gamma \to P}{\Gamma \to C}$	"C if P"	$P \Rightarrow C$
$\dfrac{\Gamma \to P' \qquad \Gamma \to P''}{\Gamma \to C}$	"C if P' and P''"	$P' \wedge P'' \Rightarrow C$
$\dfrac{\Gamma \to P^c}{\Gamma \to C}\ (c)$	"C if for all c, P"	$(\forall c.\ P) \Rightarrow C$
$\dfrac{\Gamma, A \to P}{\Gamma \to C}$	"C if whenever A, P"	$(A \Rightarrow P) \Rightarrow C$

Each rule template corresponds to a *connective* in logic and a *type constructor* in type theory

In type theory, derivations are represented mechanically by considering the object constructor/destructor corresponding to each rule template

Logics are adequate to representing *provability*

Type theories handle also *proofs*

Meta- representation

Meta-representation function

$$\ulcorner \_ \urcorner : \mathcal{L}_o \to \mathcal{L}_\mu$$

Object language | Meta-language

Judgments	$\ulcorner \_ \urcorner$	Atomic formulas (base types)
Inference rules	$\ulcorner \_ \urcorner$	Program formulas (declarations)
Deductive systems	$\ulcorner \_ \urcorner$	Static programs (signatures)
Deductions	$\ulcorner \_ \urcorner$	Terms (objects)

Adequacy theorems

– every object-level judgment/derivation has a distinct representation

– derivable judgments correspond to provable atomic formulas (inhabited types)

Approaches to meta-representation

Operational aspects of a *meta-representation system*

– verify the validity of derivations (proof-checking)

– help discovering them (proof-search)

Effective realization of proof-search

– "Theorem provers" (*NuPrl*, *ALF*, *Coq*, *Lego*, *Isabelle*)

· representation and reasoning performed in different languages

· results of proof-search are not available for meta-reasoning

· support for interactive proof development

· applicable to many meta-languages

– Logic programming languages (λ*Prolog*, *Elf*, ...)

· representation and reasoning languages are the same

· simple search strategies (e.g. resolution) are hardwired, but more complex methods can be defined

· if proofs are recorded in terms, they are available for further meta-reasoning

· no support for interactive proof development

· not generally applicable

The logical framework *LF*

Kinds	K ::= `type` | $\Pi x{:}A.\ K$
Prime types	P ::= a | P M
Types	A ::= P | $\Pi x{:}A_1.\ A_2$
Objects	M ::= x | c | $\lambda x{:}A.\ M$ | $M_1\ M_2$

Object
proof-term

Type

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

"M has type A in Γ and Σ"

Context
x:A, ...

Signature
c:A, ...
a:K, ...

Principal properties

– Type checking and type synthesis are decidable

– Can be implemented as a logic programming language (*Elf*)

– Proof-terms record the inference rules used in proving the inhabitance of a type

Meta-representation in *LF*

Judgment J	$\longrightarrow$	Base type $\ulcorner J \urcorner$
Context of J	$\longrightarrow$	Assumptions in the context Γ of *LF*
Inference rule $\dfrac{\dots}{J}$	$\longrightarrow$	Declaration r: $\ulcorner \dots \urcorner \to \ulcorner J \urcorner$
Deductive system	$\longrightarrow$	Signature Σ
Derivation for J	$\longrightarrow$	Canonical inhabitant M of $\ulcorner J \urcorner$ in Γ, Σ

Variables and binding constructs require a special treatment of the level of the syntax:

<u>Higher-order abstract syntax</u>

Adequacy theorems

$\ulcorner \_ \urcorner$ is a compositional bijection between judgments J and canonical *LF* base types $\ulcorner J \urcorner$, and between derivations $\mathcal{D}$ of J and canonical objects M such that

$$\Gamma \vdash_{\Sigma} M : \ulcorner J \urcorner$$

is derivable

Linear logic

... refines traditional logic by constraining the number of times an assumption is used in a proof.

Weakening and *contraction* are ruled out.

This calls for a finer set of connectives:

Connectives		**Context**
Traditional	T F $\neg$ $\wedge$ $\vee$ $\rightarrow$	Unbounded
Multiplicative	$\mathbf{1}$ $\perp$ ${}^{\perp}$ $\otimes$ $\wp$ $\multimap$	Split
Additive	T $\mathbf{0}$ & $\oplus$	Copy
Exponential	! ?	Unbounded

Example of sequent rules:

$$\frac{\Delta_1 \xrightarrow{ll} B \quad \Delta_2 \xrightarrow{ll} C}{\Delta_1, \Delta_2 \xrightarrow{ll} B \otimes C} \otimes L \qquad \frac{\Delta \xrightarrow{ll} B \quad \Delta \xrightarrow{ll} C}{\Delta \xrightarrow{ll} B \& C} \& L$$

Controlled forms of *weakening* and *contraction* are made availabe through the use of ! and ?

$$\frac{\Delta \xrightarrow{ll} E}{\Delta, !B \xrightarrow{ll} E} !W \qquad \frac{\Delta, !B, !B \xrightarrow{ll} E}{\Delta, !B \xrightarrow{ll} E} !C \qquad \frac{\Delta, B \xrightarrow{ll} E}{\Delta, !B \xrightarrow{ll} E} !D$$

The logical framework *LLF*

Kinds	$K ::= \texttt{type} \mid \Pi x{:}A.\ K$
Prime types	$P ::= a \mid P\ M$
Types	$A ::= P \mid \top \mid A_1 \,\&\, A_2 \mid A_1 \multimap A_2 \mid \Pi x{:}A_1.\ A_2$
Objects	$M ::= x \mid c \mid \langle\rangle \mid \langle M_1, M_2\rangle \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M$ $\mid \lambda x\hat{}A.\ M \mid M_1 \hat{}\ M_2 \mid \lambda x{:}A.\ M \mid M_1\ M_2$

Object (proof-term)

Type

$$\Psi \vdash_\Sigma M \Uparrow A$$

"M has type A in Γ and Σ"

Context
x:A, ...
x^A, ...

Signature
c:A, ...
a:K, ...

Connection to linear logic

$\Pi x{:}A.\ B \longrightarrow (!A) \multimap B$

$\Pi x{:}A.\ B \longrightarrow \forall x.\ B$

$\Gamma_1, x{:}A, \Gamma_2 \longrightarrow \Gamma'_1, !A, \Gamma'_2$

Properties of *LLF*

- Derivable terms are in η-long form

  If $\Psi \mid\!-_{\Sigma}\ U \Uparrow V$ is derivable, then Ψ, Σ, U and V are in η-long form

- Church-Rosser property

  If $U' \equiv U''$, there exists a term V such that $U' \to^{*} V$ and $U'' \to^{*} V$

- Unicity of types and kinds

  If $\Psi \mid\!-_{\Sigma}\ U \Uparrow V'$ and $\Psi \mid\!-_{\Sigma}\ U \Uparrow V''$, then $V' \equiv V''$

- Strong normalization

  If $\Psi \mid\!-_{\Sigma}\ U \Uparrow V$ is derivable, then U is strongly normalizing

- Decidability of type checking and type synthesis

  It can be recursively decided whether there exists a derivation and a term U for the judgment $\Psi \mid\!-_{\Sigma}\ U \Uparrow V$

- Convervativity over *LF*

  If Ψ, Σ, U and V do not mention linear constructs, then $\Psi \mid\!-_{\Sigma}\ U \Uparrow V$ is derivable in *LLF* iff $\Psi \mid\!-_{\Sigma}^{LF}\ U \Uparrow V$ is derivable in *LF*

Logic programming in *LLF*

Proof-search in *LLF* can be efficiently mechanized.

LLF is adequate for an implementation as a logic programming language

- Goal-directed proof-search (canonical system)
- Uniform proofs
- Resolution
- Non-determinism
  - *resource distribution*: context management
  - *conjunctive*: success continuation
  - *disjunctive*: failure continuation
  - *existential*: unification

Anatomy of a deductive system (cont'd)

Permanent context

Volatile context

$$\Gamma; \Delta \longrightarrow A$$

The volatile context needs to be handled <u>linearly</u>

$$\frac{}{\Gamma;\Delta \to C}$$
T -o C

$$\frac{}{\Gamma; \cdot \to C}$$
C

$$\frac{\Gamma;\Delta \to P}{\Gamma;\Delta \to C}$$
P -o C

$$\frac{\Gamma; \cdot \to P}{\Gamma; \cdot \to C}$$
$P \Rightarrow C$

$$\frac{\Gamma;\Delta \to P' \quad \Gamma;\Delta \to P''}{\Gamma;\Delta \to C}$$
P' & P'' -o C

$$\frac{\Gamma;\Delta_1 \to P' \quad \Gamma;\Delta_2 \to P''}{\Gamma;\Delta_1,\Delta_2 \to C}$$
P' $\otimes$ P'' -o C

$$\frac{\Gamma, A; \Delta \to P}{\Gamma;\Delta \to C}$$
$(A \Rightarrow P)$ -o C

$$\frac{\Gamma; \Delta, A \to P}{\Gamma;\Delta \to C}$$
(A -o P) -o C

$$\frac{\Gamma; \Delta \to P}{\Gamma;\Delta, A \to C}$$
A $\otimes$ P -o C

$$\frac{\Gamma; \Delta, B \to P}{\Gamma;\Delta, A \to C}$$
A $\otimes$ (B -o P) -o C

Derivations are represented as objects in a <u>linear</u> λ-calculus

Meta-representation in *LLF*

Volatile context	→	Assumptions in the linear part of the context of *LLF*
Derivations	→	Linear proof-objects

Linear binding constructs are handled by means of linear higher-order abstract syntax

LLF terms must be linearly closed

Linear parameters are treated intuitionistically

MLR: *Mini-ML* with references

$e ::= x \mid \mathbf{z} \mid \mathbf{s}\ e \mid \ldots \mid \mathbf{lam}\ x.\ e \mid e_1\ e_2 \mid \ldots \mid$
$\quad c \mid \mathbf{ref}\ e \mid \mathbf{!}e \mid \langle\rangle \mid e_1 := e_2 \mid e_1; e_2$

$\tau ::= \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \mathbf{ref} \mid \mathbf{cmd}$

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

$$\ulcorner c_1{=}v_1, \ldots, c_n{=}v_n \urcorner = \begin{array}{l} c_1{:}\mathrm{cell}, \ldots\ c_n{:}\mathrm{cell}, \\ v_1\hat{}\mathrm{contains}\ c_1 \ulcorner v_1 \urcorner, \ldots, h_n\hat{}\mathrm{contains}\ c_n \ulcorner v_n \urcorner \end{array}$$

$$\frac{K \mid\!- \mathbf{return}\ \langle\rangle \Rightarrow_{(S',c=v,S'')} a}{K \mid\!- c := v \Rightarrow_{(S',c=v',S'')} a}$$

```
ev_assign*2 : ev K ((ref C) is*2 V) A
                o- contains C V'
                o- (contains C V -o
                     ev K (return noop) A).
```

Cut-elimination for linear logic

$A ::= P \mid \top \mid A_1 \,\&\, A_2 \mid A_1 \multimap A_2 \mid A_1 \Rightarrow A_2 \mid \forall x.A$

$$\frac{\Gamma;\Delta, A_1 \rightarrow A_2}{\Gamma;\Delta \rightarrow A_1 \multimap A_2}$$

```
lolli_r: (lin A1 -o pr A2)
        -o pr (A1 lolli A2).
```

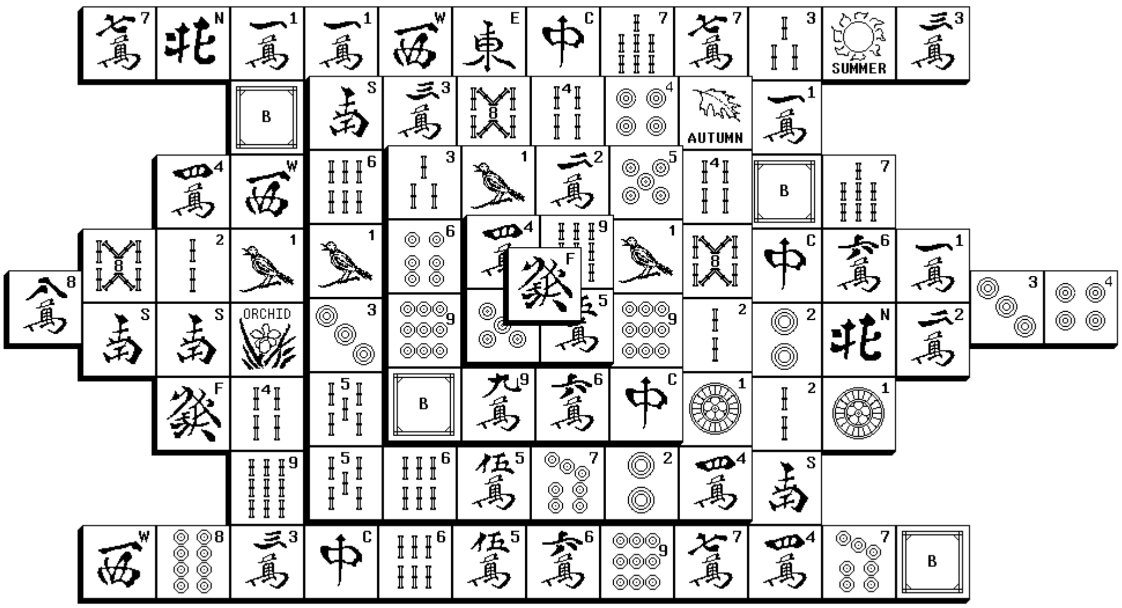
$$\frac{\Gamma;\Delta_1 \rightarrow A_1 \qquad \Gamma;\Delta_2, A_2 \rightarrow C}{\Gamma;\Delta_1, \Delta_2, A_1 \multimap A_2 \rightarrow C}$$

```
lolli_l: pr A1
        -o (lin A2 -o pr C)
        -o (lin (A1 lolli A2) -o pr C).
```

$$\frac{\Gamma;\Delta, A_1 \rightarrow A_2}{\Gamma;\Delta \rightarrow A_1 \multimap A_2} \qquad \frac{\Gamma;\Delta_1 \rightarrow A_1 \qquad \Gamma;\Delta_2, A_2 \rightarrow C}{\Gamma;\Delta_1, \Delta_2, A_1 \multimap A_2 \rightarrow C}$$

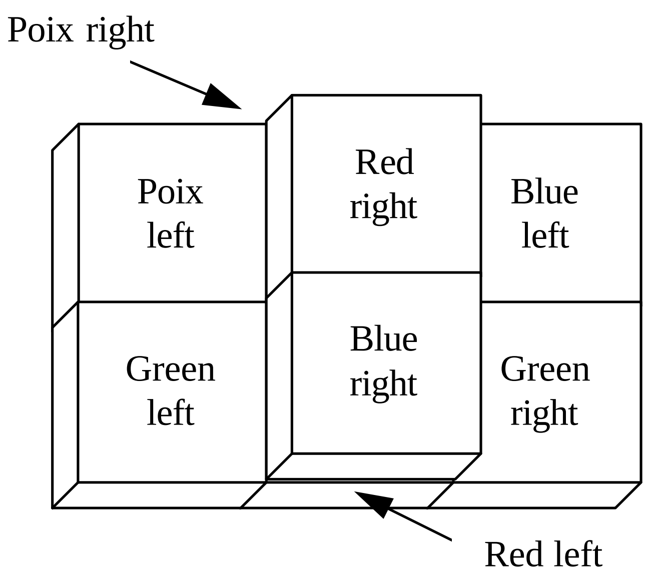
```
adm : pr A -> (lin A -> pr C)
           -> pr C -> type.

adm_lolli_r+lolli_l:
      adm (lolli_r ^ D11) (lolli_l ^ D21 ^ D22) D
         <- ({x:lin A1} adm (D11 ^ x) D22 (D' ^ x))
         <- adm D21 D' D.
```

Mahjongg




```
tp  : type.
inst: type.
tle : type.

tile: tp -> inst -> tle.

in: tle -> type.
out: tle -> type.

play: type.

match: play
         o- in (tile T N1)
         o- in (tile T N2)
         o- (out (tile T N1)
             -> out (tile T N2)
             -> play).
done : play.
```

Mahjongg (Cont'd)



green_l <-> green_r

blue_l <-> blue_r

red_l <-> red_r

poix_r <-> poix_l

```
blue: tp.     red: tp.     green: tp     poix: tp.

left: inst.                right: inst.

         ...
<linear> red_l: (in (tile red left)
                  <- out (tile blue right)
                  <- out (tile green left))
              & (in (tile red left)
                  <- out (tile blue right)
                  <- out (tile green right)).
         ...
```

```
        match ^ (<fst> green_l)       ^ (<snd> green_r) ^
[gl][gr] match ^ (<snd> blue_l)        ^ blue_r          ^
[bl][br] match ^ ((<fst> red_l) gl rb)  ^ red_r           ^
[rl][rr] match ^ ((<snd> poix_r) bl rr) ^ (<fst> poix_l)  ^
        done
```

Related work

- *λProlog*
  - representation of provability, but not of proofs
  - no treatment of linearity
- *Elf*
  - representation of provability and proofs
  - no treatment of linearity
- *Lolli, Lygon, LO*
  - limited representation of linear provability
- *Forum*
  - representation of linear provability, but not of linear proofs
- The general logic *LU*
  - polyhedric formalism: one uses the aspects he/she needs
  - no direct support for meta-representation
- Other approaches to handling state
  - monads
  - uniqueness types
  - extended DCGs

Future work

- Implementation
  - interpreter
  - compiler
  - programming environment
    - user interface (cfr. *Elf* 's Emacs mode)
    - schema checking
- Extensive library of examples
- Pure type system for *LLF*
- Definition of linear quantifiers and linear dependent types
- Long term
  - internal support for representing concurrent computations
  - treatment of I/O
  - human-oriented proof generation

Conclusions

The linear logical framework *LLF* extends the type theory of *LF* with constructs from linear logic.

– it permits representing and reasoning about:

- imperative programming languages characterized by a mutable store
- logics with a non-monotonic treatment of their context (e.g. linear logic)
- languages with linear binding constructs (linear higher-order abstract syntax)
- problems based on a state that evolves with time (e.g. puzzles and solitaires)

– is a first example of a linear type theory

– applies a proof-theoretic approach to the design of a linear logic programming language

– stands as a starting point for the study of *linear quantifiers* and *linear dependent types*