The DemoNat project

Patrick Thévenon
patrick.thevenon@univ-savoie.fr

LAMA, Université de Savoie
Le Bourget-du-Lac

31 Mars 2006
LIX, Ecole polytechnique
Palaiseau

Introduction

Introduction

- Aim of the projet :
  - Analyse and validate proofs in natural language

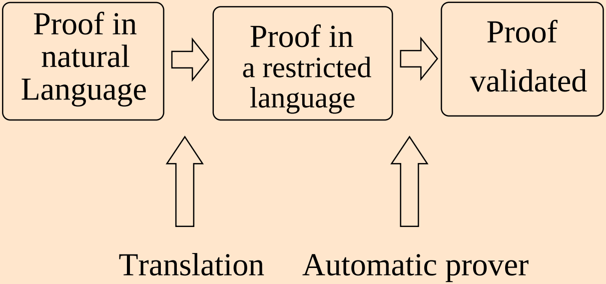
Introduction

- Aim of the projet :
  - Analyse and validate proofs in natural language
- Interest :
  - Teaching
  - Simplicity

Introduction

- Aim of the projet :
  - Analyse and validate proofs in natural language
- Interest :
  - Teaching
  - Simplicity
- Teams involved in the projet :
  - Lattice/Talana (Jussieu)
  - Calligramme (Nancy)
  - LAMA (Chambéry)

The system


Proof in natural Language
Proof in a restricted language
Proof validated
Translation
Automatic prover

My work in this project

My work in this project

- Practical :
  - Definition of a restricted language
  - Implementation of a prover

My work in this project

- Practical :
  - Definition of a restricted language
  - Implementation of a prover
- Theoretical :
  - ACGs and principal typing with two arrows
  - Study of a logic system observed from the prover

The Restricted language

The Restricted language

- Aim :
  - Describes a proof
  - Uses a small grammar
  - Allows to give hints to the prover

The Restricted language

- Aim :
  - Describes a proof
  - Uses a small grammar
  - Allows to give hints to the prover
- Features :
  - Describes a tree of logical rules
  - The grammar itself is independant from the logic

The Restricted language

- Aim :
  - Describes a proof
  - Uses a small grammar
  - Allows to give hints to the prover
- Features :
  - Describes a tree of logical rules
  - The grammar itself is independant from the logic
- Treatment :
  - Linked to a current goal
  - To each rule is associated a "trivial" goal
  - The nexts goals are given to the user

The grammar (a bit simplified) 1

nc

- ncs
- BY ... (WITH ...) ncs
- PROVE FORM nc MYIN nc
- BYABSURD HYPNAME nc
- SET EQUAL nc
- LABEL HYPNAME

ncs

- DEDUCE FORM nc
- TRIVIAL
- meta

The grammar (a bit simplified) 2

meta

LET CST meta
SEARCH VAR meta
ASSUME FORM meta
SHOW FORM nc SHOWN
meta MYTHEN meta
PBEGIN meta PEND
PROOF nc ENDPROOF

The interpretation

The interpretation

BY ... (WITH ...) : given as hints to the prover

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

DEDUCE FORM : FORM is proved

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

DEDUCE FORM : FORM is proved

TRIVIAL : the current goal is proved

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

DEDUCE FORM : FORM is proved

TRIVIAL : the current goal is proved

LET CST : a new constant added

SEARCH VAR : a new variable added

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

DEDUCE FORM : FORM is proved

TRIVIAL : the current goal is proved

LET CST : a new constant added

SEARCH VAR : a new variable added

SHOW FORM : FORM implies the current goal

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

DEDUCE FORM : FORM is proved

TRIVIAL : the current goal is proved

LET CST : a new constant added

SEARCH VAR : a new variable added

SHOW FORM : FORM implies the current goal

THEN : a new premiss for the rule

The interpretation

BY ... (WITH ...) : given as hints to the prover

PROVE FORM : the (valid) cut rule

DEDUCE FORM : FORM is proved

TRIVIAL : the current goal is proved

LET CST : a new constant added

SEARCH VAR : a new variable added

SHOW FORM : FORM implies the current goal

THEN : a new premiss for the rule

PBEGIN (...) PEND : parenthesis

PROOF (...) ENDPROOF : proof of the current premiss

The justification

The justification

- For each rule a formula is computed that justifies it
  - Share variables as much as possible
  - If no goal has changed, don't use the goal formula in the formula

The justification

- For each rule a formula is computed that justifies it
  - Share variables as much as possible
  - If no goal has changed, don't use the goal formula in the formula
- Formulas given with BY and WITH if not hypothesis
  - Are first proved
  - Are used as hints for the prover
  - Are forgotten in the next goals

The prover as a functor

The prover as a functor

```
module Prover : functor (Logic : Logic) -> 
sig
Exception Prove_fails

val prove : ( formula * int * constraints) list
                    -> formula
                    -> unit
(* raises Prove_fails when no proof is found *)
end
```

To have a prover :

- give a logic
- apply the functor to it.

Logic required

```
module type Logic =
sig
type formula (form)

val elim_all_neg : form -> form
...
type substitution (subs)

type constraints (csts)

val unif : csts -> form -> csts -> form ->
        int * subs * csts * form * form list

val get_rules : csts -> form -> bool ->
        (string * int * subs * csts * form list ) list
end
```

Resolution

- Principle : Finding a contradiction in a set of clauses (set of disjonctive formulas)
- Two rules
  - Resolution rule

$$\frac{C_1, L_1 \quad C_2, L_2 \quad \sigma = mgu(L_1, \overline{L_2})}{C_1\sigma, C_2\sigma}\, res$$

  - Contraction rule

$$\frac{C_1, L_1, L_2 \quad \sigma = mgu(L_1, L_2)}{C_1\sigma, L_1\sigma}\, contr$$

Decomposition rules 1

Decomposition rules 1

- Problem : how to compute a set of clauses from a formula ?

Decomposition rules 1

- Problem : how to compute a set of clauses from a formula ?
- We don't want to decompose everything when we have $F \longrightarrow F$ to prove

Decomposition rules 1

- Problem : how to compute a set of clauses from a formula ?
- We don't want to decompose everything when we have $F \longrightarrow F$ to prove
- The idea :
  - use decomposition rules
  - clauses are sets of formulas
    (not necessarily atomic formulas)

Decomposition rules 2

Exple : Let $\{\neg F, \Gamma\}$ be a clause with $F = (A \to B)$
From $F \leftrightarrow (A \to B)$ we obtain two clauses :
$\{A, \Gamma\}$ and $\{\neg B, \Gamma\}$
It can be seen as resolutions with the following clauses on the literal $F \equiv X_1 \to X_2$:
$\{X_1, X_1 \to X_2\}$ and $\{\neg X_2, X_1 \to X_2\}$

$\longrightarrow$ Decomposing is making resolution with rule clauses.

$\longrightarrow$ **get_rules** asks for each formula which rules can be applied.

Strategies

- Weights on each clause, computed from variables such as size of clauses, size of unifications, ...

Strategies

- Weights on each clause, computed from variables such as size of clauses, size of unifications, ...
- Deletion of subsumed clauses and tautologies

Strategies

- Weights on each clause, computed from variables such as size of clauses, size of unifications, ...
- Deletion of subsumed clauses and tautologies
- Kind of negative (positive) hyper-resolution

Strategies

- Weights on each clause, computed from variables such as size of clauses, size of unifications, ...
- Deletion of subsumed clauses and tautologies
- Kind of negative (positive) hyper-resolution
- Splitting without splitting : adding propositionnal (splitting) variables attached to clause parts in order to split clauses

Strategies

- Weights on each clause, computed from variables such as size of clauses, size of unifications, ...
- Deletion of subsumed clauses and tautologies
- Kind of negative (positive) hyper-resolution
- Splitting without splitting : adding propositionnal (splitting) variables attached to clause parts in order to split clauses

  $\longrightarrow$ OL-deduction for clauses of splitting variables

The Abstract Categorial Grammars

The Abstract Categorial Grammars

- Definition
  - Two signatures (set of typed constants)
  - a lexicon $\mathcal{L}$, morphism between the two signatures

The Abstract Categorial Grammars

- Definition
  - Two signatures (set of typed constants)
  - a lexicon $\mathcal{L}$, morphism between the two signatures
- Used for translation between :
  - abstract syntax and concrete syntax
  - abstract syntax and semantics
  - ...

The Abstract Categorial Grammars

- Definition
  - Two signatures (set of typed constants)
  - a lexicon $\mathcal{L}$, morphism between the two signatures
- Used for translation between :
  - abstract syntax and concrete syntax
  - abstract syntax and semantics
  - ...
- Condition on the lexicon :

$$\mathcal{L}(c) \ : \ \mathcal{L}(\tau(c))$$

Using ACGs

Using ACGs

- The user gives
  - The two signatures :
    1. The constants
    2. The types of the constants

Using ACGs

- The user gives
  - The two signatures :
    1. The constants
    2. The types of the constants
  - The lexicon $\mathcal{L}$:
    1. the mapping of the constants
    2. nothing more

Using ACGs

- The user gives
  - The two signatures :
    1. The constants
    2. The types of the constants
  - The lexicon $\mathcal{L}$:
    1. the mapping of the constants
    2. nothing more
- An algorithm has to :
  - find the whole lexicon (mapping on types)
  - Reverse the lexicon (not injective)

Using ACGs

- The user gives
  - The two signatures :
    1. The constants
    2. The types of the constants
  - The lexicon $\mathcal{L}$:
    1. the mapping of the constants
    2. nothing more
- An algorithm has to :
  - find the whole lexicon (mapping on types)
  - Reverse the lexicon (not injective)
- Thanks to the condition on the lexicon the mapping on types can be found thanks to a principal type algorithm

Problem

Problem

The signatures are all based on the same calculus

Problem

The signatures are all based on the same calculus

Initialy ACGs were based on linear lambda calculus

Problem

The signatures are all based on the same calculus

Initialy ACGs were based on linear lambda calculus

The linear lambda calculus, useful while dealing with syntax, is limited in its expressiveness for semantics where one needs to write formulas using several occurences of a variable

Problem

The signatures are all based on the same calculus

Initialy ACGs were based on linear lambda calculus

The linear lambda calculus, useful while dealing with syntax, is limited in its expressiveness for semantics where one needs to write formulas using several occurences of a variable

So a calculus with two kind of arrows and variables was defined

Problem

The signatures are all based on the same calculus

Initialy ACGs were based on linear lambda calculus

The linear lambda calculus, useful while dealing with syntax, is limited in its expressiveness for semantics where one needs to write formulas using several occurences of a variable

So a calculus with two kind of arrows and variables was defined

While computing a principal type some problems appear

The calculus 1

$$\frac{}{\Gamma; \vdash c : \tau(c)}$$

$$\frac{}{\Gamma; x : \gamma \vdash x : \gamma} \qquad \frac{}{\Gamma, x : \gamma; \vdash x : \gamma}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash t : \beta}{\Gamma; \Delta \vdash \lambda^{\circ} x.t : \alpha \multimap \beta} \qquad \frac{\Gamma, x : \alpha; \Delta \vdash t : \beta}{\Gamma; \Delta \vdash \lambda x.t : \alpha \rightarrow \beta}$$

The calculus 2

$$\frac{\Gamma; \Delta_1 \vdash t : \alpha \multimap \beta \quad \Gamma; \Delta_2 \vdash u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash (t\ u) : \beta}\ (*)$$

$$\frac{\Gamma; \Delta \vdash t : \alpha \rightarrow \beta \quad \Gamma; \vdash u : \alpha}{\Gamma; \Delta \vdash (t\ u) : \beta}$$

$(*)$ $Dom(\Delta_1) \cap Dom(\Delta_2) = \emptyset$

The principal typing

- We need a typing rule scheme

$$\frac{\Gamma; \Delta \vdash t : \alpha \multimap ?_n \beta \quad \Gamma; \vdash u : \alpha}{\Gamma; \Delta \vdash (t\ u) : \beta}$$

The principal typing

- We need a typing rule scheme

$$\frac{\Gamma; \Delta \vdash t : \alpha {-?_n} \beta \quad \Gamma; \vdash u : \alpha}{\Gamma; \Delta \vdash (t\ u) : \beta}$$

- usual typing algorithm (Damas-Milner) with constraints while typing application $(u\ v)$:
  - if v has free linear variables u must have type $\multimap$
  - overwise we take a new unspecified arrow $-?$ to type u

- Generally some unspecified arrows are remaining
- If we want to avoid them, there can be some problems
- Example :

  let

  $$t = \lambda g \lambda f \mathring{\lambda} x \lambda u.(g \quad (f\ x) \quad (f\ \lambda t.(t\ u)))$$

  its principal type is

  $$\begin{aligned} \vdash t\ :\ (b \multimap b {-?_1}\, n) &\quad \rightarrow \\ (((a {-?_2}\, e) \rightarrow e) \multimap b) &\quad \rightarrow \\ ((a {-?_2}\, e) \rightarrow e) &\quad \multimap \\ a &\quad \rightarrow \quad n \end{aligned}$$

  t is neither linear nor η-long

The arrow property

- a typed term has the arrow property if
  - the unspecified arrows are negative
  - the intuitionistic arrows are positive

The arrow property

- a typed term has the arrow property if
  - the unspecified arrows are negative
  - the intuitionistic arrows are positive
- linear terms have the arrow property

The arrow property

- a typed term has the arrow property if
  - the unspecified arrows are negative
  - the intuitionistic arrows are positive
- linear terms have the arrow property
- η-long terms have the arrow property

Proofs (very general ideas) 1

Proofs (very general ideas) 1

- Linear terms :
  - Generalize the property :
    1. each type variable appearing appears twice
       with a positive occurrence
       and a negative occurrence
    2. the type has the arrow property
    3. the unspecified arrows are distinct

Proofs (very general ideas) 1

- Linear terms :
  - Generalize the property :
    1. each type variable appearing appears twice
       with a positive occurrence
       and a negative occurrence
    2. the type has the arrow property
    3. the unspecified arrows are distinct
  - true for terms in β-normal form

Proofs (very general ideas) 1

- Linear terms :
  - Generalize the property :
    1. each type variable appearing appears twice with a positive occurrence and a negative occurrence
    2. the type has the arrow property
    3. the unspecified arrows are distinct
  - true for terms in β-normal form
  - it is stable under β-expansion

Proofs (very general ideas) 2

- η-long terms :
  - The typing algorithm is adapted to η-long terms

Proofs (very general ideas) 2

- η-long terms :
  - The typing algorithm is adapted to η-long terms
  - A notion of justification to each arrow and atomic type in a principal type is defined

Proofs (very general ideas) 2

- η-long terms :
  - The typing algorithm is adapted to η-long terms
  - A notion of justification to each arrow and atomic type in a principal type is defined
  - The arrow property is generalized :
    1. everything is justified (by justifying terms)
    2. if the justifying terms are variables x of $\lambda x.u$ s.t. $x \notin u$ then the type is an atom a and a is unique
    3. unspecified arrow are unique and negative
    4. the $\rightarrow$ are positive

Proofs (very general ideas) 2

- η-long terms :
  - The typing algorithm is adapted to η-long terms
  - A notion of justification to each arrow and atomic type in a principal type is defined
  - The arrow property is generalized :
    1. everything is justified (by justifying terms)
    2. if the justifying terms are variables x of $\lambda x.u$ s.t. $x \notin u$ then the type is an atom a and a is unique
    3. unspecified arrow are unique and negative
    4. the $\rightarrow$ are positive
  - If t is an η-long term with a negative $\rightarrow$ then this arrow can be replaced by a $\multimap$

Conclusion / Projects

- Practical for the prover :
  - Needs constant improvements
    functions for weight, data structures, strategies,...
  - Has been used by two classical logics
    propositional and first order
  - Will be used in PhoX, proof assistant
    developped by C. Raffalli
- Theoretical :
  - In the ACGs :
    - Work on the matching problem
      I. Cervesato defined similar calculus
    - Find another proof for the principal typing with subtypes
    - Work on another calculus, with features
  - For the prover :
    - Define a logic system to prove theoretical things
      a system between free deduction of M. Parigot
      and the calculus of structures of A. Guglielmi