# Preuve automatique de théorèmes

Scander Mustapha

*École polytechnique*

31 mai 2018

Ce rapport a pour objet de présenter l'implémentation et les résultats du projet "Automatically Proving Predicate Logic Theorems". Le code est disponible sur le repo [1]. Nous avons implémenté le stage 1, le stage 2 et l'algorithme MGU, un parser TPTP avec Menhir, les optimisations de 5.2 et la présentation des preuves en LaTeX.

## 1 Principe de l'algorithme

Soit $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ un séquent avec $(A_i)$ et $(B_j)$ des formules. On suppose que ce séquent est prouvable, et on souhaite trouver une démonstration algorithmiquement.

On suppose que notre système de preuve est l'ensemble des règles suivantes

- left-sel, right-sel
- iniL, $\Rightarrow L$, $\wedge L$, $\top L$, $\vee L$, $\bot L$
- iniR, $\Rightarrow R$, $\wedge R$, $\top R$, $\vee R$, $\bot R$
- $\forall L$, $\exists L$
- $\forall R$, $\exists R$

En appliquant ces différentes règles, on passe d'un séquent à un autre (à condition que l'application de la règle soit valide). Une preuve est alors une succesion d'application de règles et l'objectif est d'aboutir au séquent vide, que l'on peut obtenir par exemple en appliquant iniL, iniR, $\bot L$ ou $\top R$.

On peut modéliser ceci par la recherche dans un arbre : une branche correspond à l'application d'une règle, et le fils d'un noeud au séquent obtenu par l'application de la règle à ce noeud. Certaines branches de l'arbre peuvent être infinies.

Pour prouver le séquent $A_i \vdash B_j$, on construit l'arbre qui admet pour racine ce séquent, et l'on s'arrête lorsque le séquent vide est trouvé. Comme l'arbre est potentiellement infini, on impose une profondeur limite. Lorsque la profondeur limite est atteinte, on recommence en l'augmentant.

1. `https://github.com/intermet/ocaml-automatic-theorem-proof`

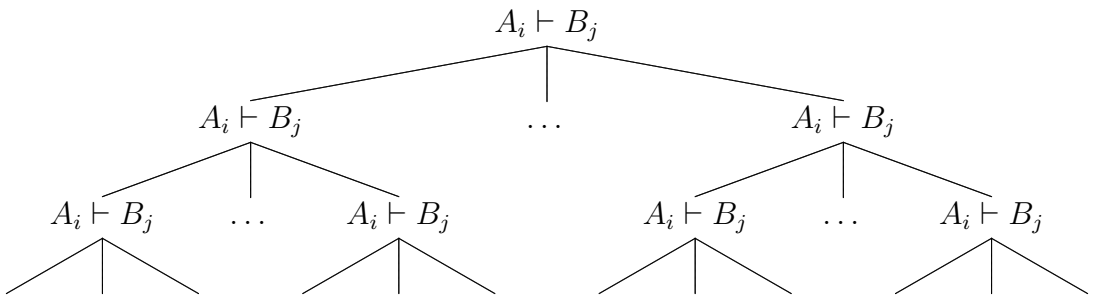


FIGURE 1 – Recherche dans un arbre

# 2 Implémentation

## Compilation

Le compilation de fait avec la commande :

```
git clone https://github.com/intermet/ocaml-automatic-theorem-proof
cd ocaml-automatic-theorem-proof
ocamlbuild -use-menhir -tag thread -use-ocamlfind main.native
```

## 2.1 Module Syntax

La signature du module est la suivante :

```
module type SYNTAX =
  sig
    type term =
      | MetaVariable of string * term list
      | Variable of string
      | Constant of string
      | Operator of string * term list

    type formula = private
      | Predicate of string * term list
      | And of formula * formula
      | True
      | Or of formula * formula
      | False
      | Implies of formula * formula
      | Forall of string * formula
      | Exists of string * formula

    exception SyntaxError

    val variable : string -> term
    val metavariable : string -> term list ->term
    val constant : string -> term
    val operator : string -> int -> term list -> term

    val predicate : string -> int -> term list -> formula
    val _and : formula -> formula -> formula
    val _true : formula
    val _or : formula -> formula -> formula
    val _false : formula
    val implies : formula -> formula -> formula
    val forall : string -> formula -> formula
    val exists : string -> formula -> formula

    val meta_replace : formula -> formula
    val const_replace : formula -> formula * term
    val add_cst_fc_f : term -> formula -> formula
    val replace_cst_var : formula -> formula

    val concatenate : string list -> string
    val format_term : term -> string
    val format_formula : formula -> string
  end
```

### Commentaires

- `MetaVariable` implémente le concept de meta-variable utilisé par le `MGU` du stage 2. Son constructeur prend un nom de variable et une liste de term. Cette liste contient les variables interdites lors de l'unification par `MGU` (cf erreur * du sujet). Une variable `Variable(x)` est transformé en meta-variable en mettant en majuscule les lettres de son nom : `MetaVariable(String.uppercase_ascii x, [])`
- `SyntaxError` est une exception levée lorsqu'une erreur de syntaxe est rencontrée.

- `meta_replace` remplace dans une formule de la forme `Forall(x, formula)` ou `Exists(x, formula)`, la variable `x` par la meta-variable associée.
- `add_cst_fc_f` ajoute une constante à la liste des constantes interdites de toutes les meta-variables d'une formule.
- `replace_cst_var` remplace les constante par des variables (cf partie du TPTP)

## 2.2 Module Kernel

La signature du modèle est la suivante :

```
module FormulaSet =
  Set.Make(
      struct
        let compare = Pervasives.compare
        type t = formula
      end
    )

module type KERNEL =
  sig
    type sequent =
      | Invalid
      | Done
      | NonSelected of FormulaSet.t * FormulaSet.t
      | SelectedL of FormulaSet.t * FormulaSet.t * formula
      | SelectedR of FormulaSet.t * FormulaSet.t * formula

    type proof =
      | None
      | SelL of formula * proof
      | SelR of formula * proof
      | IniL
      | IniR
      | ImpL of proof * proof
      | ImpR of proof
      | AndL of proof
      | AndR of proof * proof
      | TrueL of proof
      | TrueR
      | OrL of proof * proof
      | OrR of proof
      | FalseL
      | FalseR of proof
      | ForallL of proof
      | ForallR of proof
      | ExistsL of proof
      | ExistsR of proof

    val empty : FormulaSet.t
    val singleton : formula -> FormulaSet.t
    val add : formula -> FormulaSet.t -> FormulaSet.t

    val selL : sequent  -> formula -> sequent
    val selR : sequent  -> formula -> sequent

    val iniL : sequent -> sequent
    val iniR : sequent -> sequent

    val impL : sequent -> sequent * sequent
    val impR : sequent -> sequent

    val andL : sequent -> sequent
    val andR : sequent -> sequent * sequent

    val trueL : sequent -> sequent
    val trueR : sequent -> sequent

    val orL : sequent -> sequent * sequent
    val orR : sequent -> sequent

    val falseL : sequent -> sequent
    val falseR : sequent -> sequent

    val forallL : sequent -> sequent
    val forallR : sequent -> sequent

    val existsL : sequent -> sequent
    val existsR : sequent -> sequent

    val search : sequent -> int -> bool * proof

    val format_term : term -> string
    val format_formula : formula -> string
    val format_sequent : sequent -> string
    val format_proof : sequent -> proof -> string
    val format_tex : sequent -> proof -> string

    val find_proof : sequent -> proof

    val tptp_to_sequent : prop list -> sequent

  end
```

### Commentaires

- `FormulaSet` implémente un ensemble de formules. Les fonctions `empty`, `singleton` et `add` permettent respectivement de créer un ensemble vide, un singleton et d'ajouter une formule à un ensemble.
- un `sequent` est soit : un sequent `Invalid` lors de l'application d'une règle non valide, le sequent `Done` lorsque le sequent est prouvé, un sequent
  `NonSelected(left, right)` lorsque qu'aucune formule n'est sélectionnée ni à droite, ni à gauche, un sequent `SelectedL(left, right, formula)` lorsque la formule `formula` de `left` est sélectionnée à gauche, et enfin
  `SelectedR(left, right, formula)` lorsqu'elle est sélectionnée à droite. `left` et `right` sont des ensembles de formules représentant respectivement les $(A_i)$ et les $(B_j)$ dans le séquent $A_i \vdash B_j$.
- le type `proof` implémente une preuve sous la forme d'un arbre de règles.

- les différentes règles du système de preuve sont implémentées. Leur signature est assez explicite. Par exemple : `selL : sequent -> formula -> sequent` prend un séquent et une formule, et sélectionne à gauche la formule ; `orL: sequent -> sequent * sequent` prend un sequent du type `SelectedL(_, _, Or(_, _))` et renvoie les deux séquents obtenus par l'application de orL.
- la fonction `search : sequent -> int -> bool * proof` implémente la recherche du sequent `Done` dans l'arbre. La fonction est récursive : `search sequent bound` cherche *Done* en constuisant l'arbre avec une profondeur maximale égale à `bound`. Elle renvoie `(true, proof)` si une preuve a été trouvé, et `(false, None)` sinon.
- les fonctions `format_` permet de convertir un terme, une formule, un sequent ou encore une preuve en code LaTeX(cf partie correspondante).
- la fonction `find_proof` cherche une preuve en utilisant la fonction `search` en partant d'une profondeur maximale de 1 et en l'augmentant à chaque tentative de preuve.
- `tptpt_to_sequent` est relatif à la partie ...

## 2.3 Module MGU

La signature du module est la suivant :

```
module type MGU =
  sig
    type disagrement =
      | NonUnifiable
      | NoDisagrement
      | Dis of term * term

    val unifiable : formula -> formula -> bool
  end
```

### Commentaires

- ce module implémente l'algorithme Most General Unifer (MGU). Nous nous somme basé sur la présentation [2].
- `unifiable` détermine si deux formules sont unifiables.

## 2.4 Parser TPTP

Les fichiers `parser.mly` et `lexer.mll` implémente un parser du format TPTP pour les problèmes de la classe FOF. Le générateur de parser utilisé est `Menhir` [3]. Les fonctions du fichier `tptp.ml` permettent de convertir un problème fof de TPTP en un objet du type `sequent`.

2. `http://profs.sci.univr.it/~farinelli/courses/ar/slides/unification.pdf`
3. `http://gallium.inria.fr/~fpottier/menhir/`

## 2.5 Présentation des preuves en LaTeX

Nous avons utilisé le package `prftree`[4]. Par exemple :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bot, [\bot] \vdash p, (\bot \Rightarrow p)}\,\bot L}{\bot, [\bot] \vdash p, (\bot \Rightarrow p)}\,\text{left-sel}}{\bot \vdash p, (\bot \Rightarrow p)}\,\Rightarrow R}{\vdash [(\bot \Rightarrow p)], (\bot \Rightarrow p)}\,\text{right-sel}$$

$$\vdash (\bot \Rightarrow p)$$

FIGURE 2 – Preuve de $\bot \Rightarrow p$

4. `https://ctan.org/pkg/prftree`