CURIEL
Amaury

LDD3IM

RAPPORT DE PROJET
Assistant de preuve pour une logique du premier ordre sortée avec égalité

2021 – 2022

I) Spécification

A) Objectif initial

Lors de l'établissement du cahier des charges du projet, il fut décidé que le logiciel devrait pouvoir :

- Vérifier la validité d'une preuve
- Refuser une preuve fausse
- Informer l'utilisateur quand une tactique a échoué et expliquer pourquoi
- Assister l'utilisateur dans la démonstration de toute formule du 1er ordre
- L'utilisateur pourra définir de façon simple la signature utilisée
- L'utilisateur pourra introduire de façon simple des définitions : par exemple pouvoir remplacer $\forall \varepsilon > 0, \exists \delta > 0, \forall x, |x - x_0| \leq \delta \rightarrow |f\ x - f\ x_0| \leq \varepsilon$ par « continue(f,x0) »
- Intégrer les opérations de bases (égalité, +, -, *, ...)
- Automatiser certains raisonnements

B) Solution technique

Dans cette partie, nous allons détailler point par point comment nous avons réussis à implémenter les fonctionnalités de la partie A) :

- Vérifier la validité d'une preuve :
  Lorsque l'on prouve un théorème par le biais d'un assistant de preuve, la preuve est valide à condition qu'il n'y ait pas de bugs dans l'assistant de preuve. Il est donc crucial de s'assurer du fait qu'aucun bug ne soit présent dans l'assistant de preuve. Pour ce faire, des tests unitaires testant chacune des tactiques ont été implémentés. De plus, lorsque l'on essaye d'appliquer une tactique inapplicable, une erreur est renvoyée expliquant pourquoi la tactique a échoué.
- Assister l'utilisateur dans la démonstration de toute formule du premier ordre :
  Lorsque l'on démontre un gros théorème, les formules utilisées peuvent s'avérer très complexe ainsi, il peut être compliquer de sélectionner exactement la bonne tactique à utiliser. Pour pallier cela, le logiciel permet de simplement choisir le type de tactique à appliquer : il suffit par exemple de taper « INTRO » suivis des éventuels arguments pour utiliser une tactique d'introduction.
- L'utilisateur pourra définir de façon simple la signature utilisée :
  Pour faire une démonstration, il est nécessaire de pouvoir définir la théorie utilisée. Pour permettre cela, le logiciel va lire le fichier contenant la signature (ex : exemple4.txt) afin de la charger en mémoire et de l'analyser pour s'assurer qu'elle est bien correcte (pas d'applications mal formées par exemple).
- L'utilisateur pourra introduire de nouvelles définitions :
  En plus d'être long, avoir des formules trop longues est également source d'erreur. C'est pour cela que le logiciel propose de pouvoir ajouter à la signature une définition de formule pour pouvoir plus tard écrire seulement le nom de la propriété au lieu de la propriété en formule logique.
- Intégrer les opérations de bases :
  Pour ce point-ci, seul l'égalité a été implémentée.
- Les autres points n'ont pas eu le temps d'être implémentés

II) Conception

A) Résumé de la syntaxe

1) Partie non interactive

a) Signature

Pour rédiger une preuve avec le logiciel, il faudra tout d'abord renseigner la théorie utilisée. Pour définir le langage utilisé, il suffit de rentrer le mot clef SIGMA suivis d'entre accolade du type du nom et de la sorte des symboles utilisés. Ces déclarations doivent être séparés par des points-virgules. Par exemple un langage pour la théorie des groupes pourrait être déclaré de la manière suivante :

SIGMA {sort g; const e: g; fun star: g*g->g; fun phi: g -> g ;}

b) Axiomes & hypothèses

Une fois la signature renseignée, il faut maintenant préciser le système d'axiomes que l'on va utiliser pour montrer notre théorème. Pour ce faire, il faudra utiliser le mot clef HYP suivi du nom de l'hypothèse/axiome de deux points puis de la formule de l'axiome par exemple pour supposer que chaque élément d'un groupe possède un inverse il faudra rentrer

HYP AX1 : forall(x:g), exists(y:g), star(x,y) = e

c) Formules

Un autre élément clef de la syntaxe sont les formules : Les mots clefs sont :

Mot clef	Connecteur correspondant	Exemple
forall(nomVar1 : sorte1 ; … ;nomVarN : sorteN),formule	$\forall x1 \in ens; \dots; xn \in ens, formule$	forall(x:g;y:g), phi(star(x,y)) = star(phi(x),phi(y))
exists(nomVar1 :sorte1 ;… ;nomVarN :sorteN),formule	$\exists x1 \in ens1; \dots; xn \in ens2, formule$	Forall(x :g), exists(y:g), star(x,y) = e
formule1 and formule2	$formule1 \wedge$ formule2	surjective(F) and surjective(G)
formule1 => formule2	$formule1 => formule2$	surjective(F) and surjective(G) => surjective(comp(F,G))
formule1 or formule2	$formule1 \vee$ formule2	surjective(F) or surjective(G)
not formule	¬formule	not surjective(g)
TRUE	⊤	TRUE
FALSE	⊥	FALSE
t1 = t2	$T1 = T2$	5.8=5.8

d) Définitions

Le logiciel offre la possibilité de stocker des définitions pour cela, il suffit de rentrer le mot clef DEF, le nom de la définition, entre parenthèse les arguments (constantes seulement), deux points le mot clef prop le symbole := et la formule par exemple pour dire qu'une fonction h est surjective, on va entrer :

DEF surjective (h:freal): prop := forall(y:real), exists(x:real), app(h,x)= y

e) Objectif

Pour rentrer un objectif, il suffit de reprendre la syntaxe donnée dans b) en remplaçant le mot clef HYP par le mot clef GOAL. Par exemple :

GOAL G1: surjective(F) and surjective(G) => surjective(comp(F,G))

2) Partie interactive

a) Les tactiques

Le logiciel reconnait 3 types de tactiques : les tactiques d'introduction, d'élimination et les tactiques dérivées.

Pour exécuter une de ces tactiques, il est nécessaire d'écrire le mot clef correspondant : ELIM pour une technique d'élimination, INTRO pour une tactique d'introduction et DERIVE pour une tactique dérivée. Lorsque la tactique attend des arguments, il suffit de donner ces arguments en les séparant par des points virgule. Lorsqu'une tactique attend un nouveau nom, il est nécessaire de l'entourer de guillemets

Par exemple dans le fichier exemple 2, pour réaliser l'introduction de l'implication il faut écrire INTRO. Puis pour séparer le « et » de H_3 il faut écrire : DERIVE : H_3

Puis, pour introduire la variable proposée par G1, il faut rentrer INTRO : « z »

Enfin pour utiliser z dans H_6 on peut rentrer ELIM : H_6 ; z ou encore ELIM :0 ; z (on peut aussi rentrer le numéro de l'hypothèse (numérotées de haut en bas en partant de 0)).

La tactique $\frac{t1=t2\in\Gamma \qquad \Gamma:\mathrm{p(t1)}}{p(t1)[t1<-t2]}$ est également implémentée. Pour l'utiliser, il est nécessaire de rentrer le mot clef REPLACE suivi du nom de l'hypothèse contenant l'égalité ainsi que du nom de l'hypothèse contenant la formule que l'on veut remplacer.

b) Autre commande

L'implémentation du logiciel nous impose de travailler uniquement sur le premier séquent. Ainsi, une commande SWAP permets de mettre le séquent désiré en première position. Par exemple si nous avons 3 séquents, SWAP : n va mettre le n+1ème séquent en première position.

Une autre fonctionnalité implémentée est la possibilité de sauvegarder un état de preuve pour y revenir plus tard (dans la meme session !). Ainsi, la commande SAVE permet cela. Il est ensuite possible d'entrer un nouveau théorème (sous la même signature). On peut revenir à tout moment a la preuve initiale avec le mot clef RESUME. Il n'est possible de sauvegarder qu'une seule preuve en meme temps.

B) Fonctionnement interne

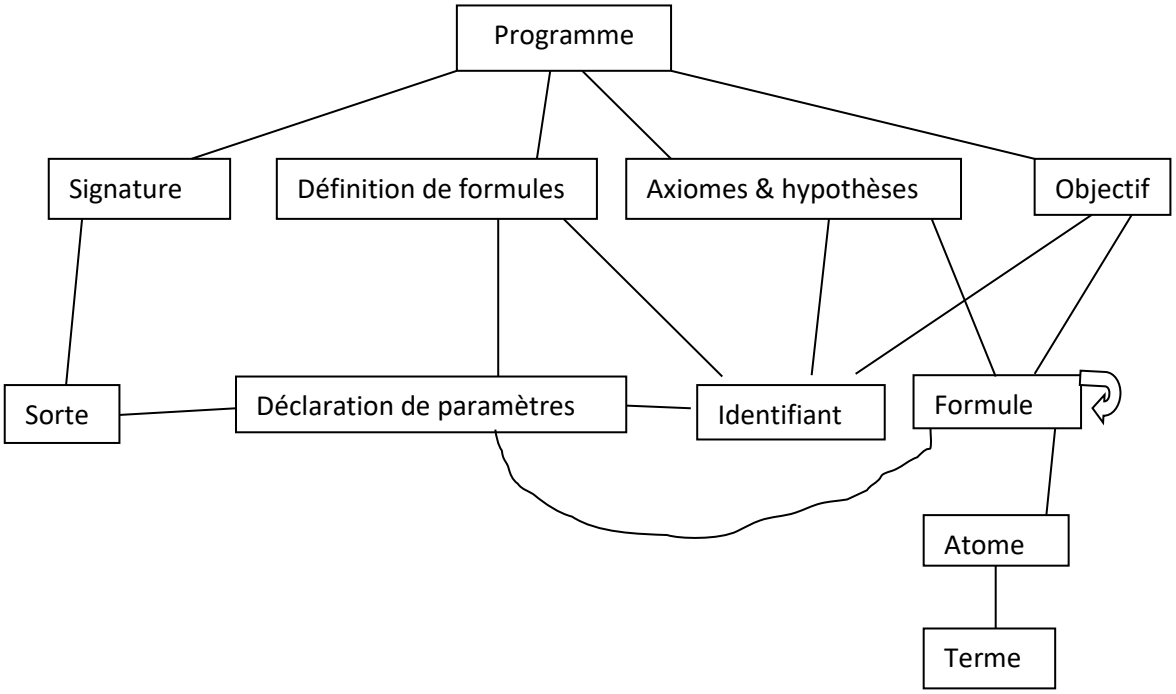
1) Partie statique

Lorsque l'utilisateur a rentré la théorie dans laquelle il souhaite travailler, le logiciel doit reconnaitre ce que l'utilisateur a entré et nous assurer que ce qu'il a entré est correct. Pour ce faire, nous avons implémenter un lexeur et un parseur.

Le lexeur vas lire le fichier entré par l'utilisateur et transformer une suite de caractères en une suite de « mots » appelés TOKENS. Les TOKENS vont ensuite être transmis au parseur qui va s'assurer que les « phrases » de TOKENS sont bien formées et, transformer ces phrases en arbres de syntaxe abstraite (abrégé plus tard en AST pour Abstract Syntaxe Tree).

L'AST est définis dans le fichier proof_ast.ml et considère un programme comme étant une signature (vide ou non), une liste de définitions (vide ou non), une liste d'hypothèses/axiomes (vide ou non), et un objectif (non vide). La signature est composée de déclarations de sortes, de relations, de constantes et de fonctions, Elles-mêmes composées d'identifiants et de sortes.

L'intégralité de l'AST est disponible dans le fichier proof_ast. On ne va ici seulement représenter un schéma simplifié de l'AST.



Une fois l'AST construit, il faut s'assurer que ce qui a été rentrer par l'utilisateur a bien un sens il faut donc analyser l'AST. Pour cela, nous avons construit une table des symboles représentée par une table de hachage contenant tout les symboles statiques renseignés par l'utilisateur. Cette table nous permet de savoir par exemple si l'utilisateur rentre « surjective(f) » si surjective est une fonction ou une définition qu'il faut appliquer. Ensuite, il faut s'assurer que l'utilisateur écrit quelque chose de juste pour cela, nous avons écrit des fonctions chargées d'analyser les formules. En particulier, la fonction analyse_form vas s'assurer que la formule analysée ne subis pas de capture et que l'on n'a pas appliqué de fonction à un therme de la mauvaise sorte.

2) Fonctionnement des tactiques

Une fois que le fichier à été analysé avec succès, l'utilisateur peux maintenant commencer sa démonstration. Pour cela, il doit rentrer des mots clefs qui seront parsés de la même manière que la signature puis déclencheront la tactique appropriée. Pour cela, nous avons créer un type Sequent pour représenter l'état de la preuve. Le séquent initial est créé en mettant toutes les hypothèses dans la catégorie hypot, et l'objectif dans la partie théorème. Une fois le séquent initial créer, on peut lui appliquer une tactique. Une tactique prend en argument un séquent et renvoie une liste de séquent. Cela continue jusqu'à que la preuve soit terminée.

```
type sequent =

  { var : (string * sort ) list;
    hypot: (string* form) list;
    theoreme: (string*form) ;
    thi: (string * form );
    axiome: (string * form) list;

  }
```

Type séquent

```
let et seq =
  let na,fo = seq.theoreme in
  match fo with
    |And(f1,f2) -> let sl =  [{var = seq.var; hypot = seq.hypot;theoreme =
(na,f1);thi = seq.thi; axiome = seq.axiome};{var = seq.var; hypot =
seq.hypot;theoreme = (na,f2);thi = seq.thi; axiome = seq.axiome}] in
print_string("success"); sl
    |_->raise  (PreuveInvalide("et non applicable"))
```

Tactique $\frac{\Gamma:A \qquad \Gamma:B}{\Gamma:A \wedge B}$

III) Eléments quantitatifs & exemples

A) Eléments quantitatifs

Le projet à commencer le 17 janvier pour se terminer le 15 mai. Les trois premières semaines ont été consacrées à la définition du sujet, les semaines du 10 février au 11 mars ont été consacrées à l'écriture du parseur et à la partie d'analyse. Du 11 mars au 6 mai, nous avons créer le langage de tactique et le reste du temps a été consacré à la gestion des conflits, a la résolution des bugs, et à l'écriture des tests unitaires.
Le projet contient 8 fichiers de code pour un total d'environ 1282 lignes de code

B) Démonstration sur un exemple

Dans cette partie, nous allons illustrer le fonctionnement du logiciel par un exemple. (Fichier exemple5.txt)
L'écran de départ :

----------HYPOTHESES & AXIOMES----------
----------OBJECTIF----------
G1 ((∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y)) ⇒ ∀ _yreal , ∃ _x , (f(g(_x)) = _y))

Pour commencer, on souhaite introduire l'implication : on tape *INTRO* :

----------HYPOTHESES & AXIOMES----------
H_0 (∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y))
----------OBJECTIF----------
G1 ∀ _yreal , ∃ _x , (f(g(_x)) = _y)

On souhaite se défaire du ∧ dans H_0 : on tape *DERIVE :H_0 :*
----------HYPOTHESES & AXIOMES----------
H_2 ∀ _yreal , ∃ _x , (f(_x) = _y)
H_1 ∀ _yreal , ∃ _x , (g(_x) = _y)
H_0 (∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y))
----------OBJECTIF----------
G1 ∀ _yreal , ∃ _x , (f(g(_x)) = _y)

On introduit ensuite le z : *INTRO : « z »*
----------HYPOTHESES & AXIOMES----------
H_2 ∀ _yreal , ∃ _x , (f(_x) = _y)
H_1 ∀ _yreal , ∃ _x , (g(_x) = _y)
H_0 (∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y))
z real
----------OBJECTIF----------
G1 ∃ _x , (f(g(_x)) = z)
On spécifie ensuite H_2 avec z : *ELIM :H_2 ;z*

----------HYPOTHESES & AXIOMES----------
H_3 ∃ _x , (f(_x) = z)
H_2 ∀ _yreal , ∃ _x , (f(_x) = _y)
H_1 ∀ _yreal , ∃ _x , (g(_x) = _y)
H_0 (∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y))
z real
----------OBJECTIF----------
G1 ∃ _x , (f(g(_x)) = z)
On récupère ensuite le x de H_3 *: ELIM : H_3 ;« y »* :
----------HYPOTHESES & AXIOMES----------
H_4 (f(y) = z)
H_3 ∃ _x , (f(_x) = z)
H_2 ∀ _yreal , ∃ _x , (f(_x) = _y)
H_1 ∀ _yreal , ∃ _x , (g(_x) = _y)
H_0 (∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y))
y real
z real
----------OBJECTIF----------
G1 ∃ _x , (f(g(_x)) = z)
On spécifie H1 avec y et on récupère une nouvelle variable x : *ELIM : H_1 ; y* puis *ELIM : 0 ; « x »* :

----------HYPOTHESES & AXIOMES----------
H_6 (g(x) = y)
H_5 ∃ _x , (g(_x) = y)
H_4 (f(y) = z)
H_3 ∃ _x , (f(_x) = z)
H_2 ∀ _yreal , ∃ _x , (f(_x) = _y)
H_1 ∀ _yreal , ∃ _x , (g(_x) = _y)
H_0 (∀ _yreal , ∃ _x , (f(_x) = _y) ∧ ∀ _yreal , ∃ _x , (g(_x) = _y))
x real
y real
z real
----------OBJECTIF----------
G1 ∃ _x , (f(g(_x)) = z)

On introduit le il existe avec x : *INTRO :x*
----------OBJECTIF----------
G1 (f(g(x)) = z)
Puis on remplace g(x) par y : *REPLACE : 0*
----------OBJECTIF----------
G1 (f(y) = z)
C'est exactement l'hypothèse H_4 :
EXACT
Ce qui conclus la preuve.

IV) Conclusion

Pour conclure, le logiciel qui a été développé dans le cadre de ce projet est entièrement fonctionnel et peut être utilisé dans le cadre d'une démonstration d'un théorème. Cependant, le fait que l'on soit au premier ordre crée certaines contraintes comme le fait que l'on ne puisse pas quantifier sur des fonctions. Ce qui nous oblige à multiplier les déclarations pour pouvoir démontrer des résultats d'analyse (en particulier pour l'utilisation de définition). Il pourrait être envisageable d'améliorer le logiciel en faisant en sorte de pouvoir simplifier l'application de définition, en ajoutant la possibilité de pouvoir sauvegarder sa preuve pour y revenir plus tard (dans une autre session) ou de manière plus ambitieuse, de pouvoir automatiser un raisonnement.