Projet MIDL 1

Implémentation de la procédure de décision dans la théorie des Ordres Denses

DAVID--MULLER Robin

MARCHAND Nicolas

VASSEUR-LAURENS Odin

Aperçu du Projet

Contenu :

Détails du projet :

- **UE :** Projet MIDL 1
- **Date de rendu :** 09/02/2026
- **Langage :** OCaml

Lot 1

Ce projet s'inscrit dans le cadre de la vérification de formules logiques pour la théorie des ordres denses sans bornes.

Le problème central est de déterminer la vérité d'une formule mathématique contenant, dans un premier temps, des comparaisons et des quantificateurs. La difficulté dans la vérification de ces formules réside dans le fait que ces quantificateurs portent sur des domaines « infinis », ne permettant pas de vérifier naïvement chaque solution par un algorithme dit de « bruteforce ». Nous résolvons donc ce problème par la manipulation algébrique des formules, permettant une élimination systématique des quantificateurs.

Suite à l'approbation de nos professeurs, nous avons choisi de développer notre projet dans le langage OCaml, contrairement aux recommandations initiales d'utiliser Python. Il y a deux raisons à cela :

- La représentation algébrique d'une formule se fait beaucoup plus naturellement sur OCaml, grâce au principe mathématique de « types inductifs », naturellement introduits par OCaml. Ceux-ci, combinés au très puissant « pattern matching », nous permettent d'obtenir un code bien plus lisible et court qu'en Python.
- Le troisième semestre de MIDL nous a introduit à l'UE ILU1. Nous y avons appris les bases d'OCaml, et ce projet nous a permis de nous familiariser avec ce langage dans un cadre approprié ainsi que d'en apprendre certaines fonctionnalités telles que les types inductifs et le pipelining.

Nous avons donc commencé par traduire le script python fourni par nos professeurs dans notre langage de choix.

Ensuite, nous avons assez facilement implémenté le prétraitement. Notre choix a été de séparer chaque clause de notre forme normale disjonctive dans une liste après avoir poussé les quantificateurs existentiels dans celle-ci.

Nous voilà donc en train de traiter un ensemble de disjonctions. Nous avons donc choisi de classifier dans un dictionnaire toutes les contraintes sur chaque variable quantifiée : égalités, majorantions, minorations, formules indépendantes (χ). Maintenant que cela est fait, nous avons appliqué toutes les règles de suppression de quantification universelle, à l'exception de la seconde (« si ψ contient $x < x$: $(\exists x.\psi) \leftrightarrow \bot$ ») que nous avons supprimé lors de notre implémentation du Lot 2 ; cela sera donc détaillé plus tard.

Lot 2

Maintenant que nous pouvons appliquer notre procédure à toute proposition de la théorie des ordres denses, nous pouvons nous pencher sur l'arithmétique des nombres rationnels. On remarque que peu de choses changent, il n'y a pas de collision entre le prétraitement et la complexification des (in)égalités « générales » qui se résumaient à des comparaisons entre deux variables.

Ce qui change le plus, c'est que l'on ne peut pas simplement avoir de majoration/minoration sur une variable x en regardant le terme de gauche ou le terme de droite. Dans le cas d'une égalité, c'est facile ! Nous avons appris la méthode du pivot de Gauss qui convient parfaitement à cette situation, il nous suffira de l'appliquer à notre égalité pour isoler x. Dans le cas d'une inégalité, nous appliquons la même chose, la seule différence est qu'il faut faire attention aux changements de signe (c'est ce que l'on appelle l'« Élimination de Fourier-Motzkin »). Ainsi, nous obtenons des majorations/minorations de x par rapport aux autres variables.

Supposons l'inégalité : $A < B$. Dans ce cas, celle-ci est équivalente à $A - B < 0$. Nous pouvons effectuer une extraction du coefficient x dans $A - B$ (cela pourrait s'apparenter à une sorte de division euclidienne d'un polynôme de degré 1 par x). Ainsi, nous aurons un coefficient scalaire q et un reste R ne dépendant pas de x. Cela nous donnera : $qx + R < 0 \iff qx < -R$.
- Dans le cas où $q = 0$, nous avons une proposition indépendante de x : $R < 0$.
- Dans le cas où $q > 0$, nous avons $x < -\frac{R}{q}$
- Dans le cas où $q < 0$, nous avons $x > -\frac{R}{q}$

Le cas de l'égalité s'effectue de la même manière.

C'est cette étape de normalisation qui rend l'étape 3.2 mentionnée précedemment (suppression triviale de $x < x$) obsolète : notre algorithme transforme naturellement $x < x$ en $0 < 0$, qui est identifié comme une formule indépendante (fausse) et traitée lors de l'étape 3.6.

Nous avons d'abord introduit une fonction `simplify_term` qui simplifiera au maximum les formules algébriques, notamment à l'aide de la règle d'associativité, pour simplifier les calculs. Ensuite, nous avons créé la fonction d'extraction de coefficients `get_coeff` pour finalement l'appliquer à la fonction de normalisation `isolate`.

Il a aussi fallu changer les fonctions implémentées précédemment : après avoir supprimé l'étape 3.2, nous avons rajouté les comparaisons arithmétiques à la fonction `simplify`. Nous pensions avoir fini lorsque l'exemple de proposition « $\exists x.x + 5 > x + 2$ » ne fonctionnait pas. Le dysfonctionnement venait de la fonction `check_var` car elle ne devait plus seulement regarder à droite et à gauche de l'inégalité, mais bien parcourir récursivement l'arbre syntaxique créé par les expressions arithmétiques à gauche et à droite. Ainsi, après avoir fait ce changement, notre procédure s'est mise à fonctionner sur tous nos tests.

Il nous aurait été possible de traiter des fractions, cela nous aurait permis de coller avec

Manuel

Pour lancer le code dans un REPL, vous pouvez exécuter la commande : `./run_repl.sh`. Sinon, vous pouvez lancer le REPL manuellement en exécutant `utop -init syntax.ml`

Construction d'une formule

Nous avons créé des fonctions macro permettant une construction assez simple des propositions. Voici un tableau les regroupant :

Constructeur logique	Équivalent
$\top$	`top`
$\bot$	`bottom`
$x < y$	`lt (var "x") (var "y")`
$x = y$	`equal (var "x") (var "y")`
$P \vee Q$	`disj P Q`
$P \wedge Q$	`conj P Q`
$\neg P$	`notf P`
$P \to Q$	`implies P Q`
$\exists x.P$	`exists "x" P`
$\forall x.P$	`forall "x" P`
x (variable)	`var x`
n (nombre)	`val_ n`
$x + y$	`add x y`
$x - y$	`sub x y`
$x \times y$	`mul x y`

Exemples

- $\forall x.\forall y.(x < y \to \exists z.(x < z \wedge z < y))$ s'écrira :

```
let density = forall "x" (
  forall "y" (
    implies (
        lt (var "x") (var "y")
      )
      (
        exists "z" (
          conj (
              lt (var "x") (var "z")
            )
            (
              lt (var "z") (var "y")
            )
        )
      )
    )
  )
```

- $\forall x.\exists y.(x < y)$ s'écrira :

```
let no_sup_extremum = forall "x" (exists "y" (lt (var "x") (var "y")))
```

L'exécution de la procédure se fera via la fonction `final_test` avec le nom de la formule. Exemple :

```
final_test density "Densité"
```

Qui renverra :

```
Nom de la formule : "Densité"
Input initial : (∀x.(∀y.(¬(x < y) ∨ (∃z.((x < z) ∧ (z < y))))))
Output final : T
```