TP9 : Logique

26 janvier et 02 février

Dans tout le sujet, on se donne $\mathcal{V}$ un ensemble infini dénombrable de variables.

On a vu en cours un exemple de système de preuve correct et non complet (qu'il n'est pas compliqué de rendre complet) pour la logique propositionnelle : la **déduction naturelle**. Le défaut de ce système de preuve est qu'il n'est pas très pratique à utiliser car il n'existe pas de stratégie de preuve simple pour chercher un arbre de preuve. Ainsi, il est difficile de systématiser la construction des preuves des exemples du cours.

1 Calcul des séquents

Le but de cette première partie est d'étudier un autre système de preuve correct et complet, mais possédant une stratégie de recherche de preuve simple : le **calcul des séquents**.

Dans la déduction naturelle, un séquent est de la forme $\Gamma \vdash \varphi$: on ne peut avoir qu'une seule formule dans la partie droite du séquent. Dans le cadre du calcul des séquents, on se donne une définition plus large de séquent : dans ce devoir, un séquent est un un couple de deux ensembles finis Γ et Δ de formules, noté $\Gamma \vdash \Delta$. Un tel séquent est dit valide (noté $\Gamma \models \Delta$) si pour toute valuation ν de $\mathcal{V}$, si ν satisfait toutes les formules de Γ, alors **il existe** une formule de Δ qui est satisfaite par ν.

Remarque : si Δ ne contient qu'une seule formule, cette définition de séquent valide coïncide avec celle du cours.

L'intérêt d'autoriser plusieurs formules dans la partie droite du séquent est d'obtenir des règles d'inférences plus "symétriques".

Dans la déduction naturelle, il y a deux types de règles :

* les règles d'**introduction** qui permettent de traiter directement la formule à droite du séquent ;
* les règles d'**élimination** qui permettent en fait de gérer indirectement une formule du contexte Γ.

Le défaut des règles d'élimination (pour pouvoir automatiser la recherche de preuve) est qu'il faut deviner quelle nouvelle formule faire apparaitre dans les prémisses. Plus généralement, il faut deviner à quel moment utiliser quelle règle, car appliquer une règle au "mauvais" moment risque de faire apparaitre une prémisse non prouvable, alors que le séquent initial était prouvable.

Dans le calcul des séquents, les règles (présentées ci-dessous) sont plus simples, et ces deux problèmes n'existent pas. Il n'y a que des règles d'introduction : celles permettant de gérer une formule de Δ, et celles permettant de traiter une formule de Γ.

$$\frac{}{\Gamma, \varphi \vdash \varphi, \Delta}\ (Ax)$$

$$\frac{}{\Gamma, \bot \vdash \Delta}\ (\bot) \qquad\qquad \frac{}{\Gamma \vdash \top, \Delta}\ (\top)$$

$$\frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}\ (\wedge \vdash) \qquad\qquad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}\ (\vdash \wedge)$$

$$\frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}\ (\vee \vdash) \qquad\qquad \frac{\Gamma \vdash \varphi, \psi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta}\ (\vdash \vee)$$

$$\frac{\Gamma \vdash \varphi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \to \psi \vdash \Delta}\ (\to \vdash) \qquad\qquad \frac{\Gamma, \varphi \vdash \psi, \Delta}{\Gamma \vdash \varphi \to \psi, \Delta}\ (\vdash \to)$$

$$\frac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta}\ (\neg \vdash) \qquad\qquad \frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}\ (\vdash \neg)$$

1.1 Quelques exemples

On commence par prouver quelques formules à l'aide de ce nouveau système de preuve.

1. Prouver $\vdash p \vee \neg p$ avec le calcul des séquents.
2. Prouver $\vdash ((p \to q) \to p) \to p$ avec le calcul des séquents.
3. Prouver $\neg(p \vee q) \vdash \neg p \wedge \neg q$ avec le calcul des séquents.
4. Prouver $\neg p \wedge \neg q \vdash \neg(p \vee q)$ avec le calcul des séquents.

1.2 Implémentation du système de preuve via le calcul des séquents.

La preuve de complétude (que l'on verra en TD) est constructive : elle nous permet d'en déduire un algorithme décidant si un séquent est valide ou non. On se propose ici de l'implémenter en OCaml.

On se donne le type `'a prop` suivant :

```
type 'a prop =
  | Top
  | Bot
  | V of 'a
  | Not of 'a prop
  | And of 'a prop * 'a prop
  | Or of 'a prop * 'a prop
  | Impl of 'a prop * 'a prop
;;
```

Dans la suite, il sera plus pratique de séparer Γ (resp. Δ) en deux parties : celle ne contenant que des variables, $\top$ ou $\bot$; et celle contenant les autres formules de Γ (resp. Δ). On se donne donc le type `'a sequent` suivant :

```
type 'a sequent = {
  gamma : 'a prop list ;
  delta : 'a prop list ;
  gamma_var : 'a prop list ;
  delta_var : 'a prop list
}
;;
```

1. Écrire une fonction `create_sequent : 'a prop list -> 'a prop list -> 'a sequent` telle que `create_se` renvoie un `'a sequent` dont les champs `gamma_var` et `delta_var` sont des listes vides, et les champs `gamma` et `delta` contiennent les listes passées en arguments.
2. Écrire une fonction `member : 'a -> 'a list -> bool` qui teste si un élément est dans une liste.
3. Écrire une fonction `bot : 'a sequent -> bool` qui teste si la règle $(\bot)$ peut être appliquée au séquent pris en argument.
   Attention : il faut chercher dans `gamma` **et** dans `gamma_var`.
4. Écrire une fonction `top : 'a sequent -> bool` qui teste si la règle $(\top)$ peut être appliquée au séquent pris en argument.
   Attention : il faut chercher dans `delta` **et** dans `delta_var`.
5. Écrire un fonction `axiom : 'a sequent -> bool` qui teste si la règle (Ax) est applicable au séquent pris en argument.
   Attention : la formule à trouver peut être dans `gamma` ou `gamma_var`, et dans `delta` ou `delta_var`.

La stratégie de preuve proposée ici est très simple :
— Si on peut appliquer (Ax) ou $(\bot)$ ou $(\top)$, on l'applique et on a prouvé le séquent.
— Sinon :
  - si `gamma` n'est pas vide, on regarde la première formule de la liste :
    * si c'est une variable, $\bot$ ou $\top$, on l'enlève de `gamma` et on le rajoute dans `gamma_var` ;
    * sinon, c'est une formule ayant un connecteur logique : on applique la règle correspondante, et on continue la recherche de preuve sur la ou les prémisses ;
  - sinon (`gamma` est vide), on procède de manière similaire avec `delta`.
    * Si `gamma` et `delta` sont vides, et que les règles (Ax), $(\bot)$ et $(\top)$ ne s'appliquent pas, alors aucune règle ne s'applique, et le séquent n'est pas valide.

On commence par implémenter chacune des règles par une fonction OCaml. En accord avec la stratégie présentée ci-dessus, si la première formule de `gamma` (resp. `delta`) n'est pas celle sur laquelle on peut appliquée la règle considérée, on lèvera l'exception suivante :

```
exception Wrong_rule of string ;;
```

6. Écrire une fonction `and_gamma : 'a sequent -> 'a sequent` qui renvoie la prémisse de la règle $(\wedge \vdash)$ appliquée à la première formule du champ `gamma` du séquent pris en argument.
   On lèvera l'exception `Wrong_rule "And Gamma"` si cette formule n'est pas une conjonction.
7. Écrire une fonction `or_gamma : 'a sequent -> 'a sequent * 'a sequent` qui renvoie les prémisses de la règle $(\vee \vdash)$ appliquée à la première formule du champ `gamma` du séquent pris en argument.
   On lèvera l'exception `Wrong_rule "Or Gamma"` si cette formule n'est pas une disjonction.
8. Écrire une fonction `impl_gamma : 'a sequent -> 'a sequent * 'a sequent` qui renvoie les prémisses de la règle $(\rightarrow \vdash)$ appliquée à la première formule du champ `gamma` du séquent pris en argument.
   On lèvera l'exception `Wrong_rule "Impl Gamma"` si cette formule n'est pas une implication.
9. Écrire une fonction `not_gamma : 'a sequent -> 'a sequent` qui renvoie la prémisse de la règle $(\neg \vdash)$ appliquée à la première formule du champ `gamma` du séquent pris en argument.
   On lèvera l'exception `Wrong_rule "Not Gamma"` si cette formule n'est pas une négation.
10. Écrire une fonction `and_delta : 'a sequent -> 'a sequent * 'a sequent` qui renvoie les prémisses de la règle $(\vdash \wedge)$ appliquée à la première formule du champ `delta` du séquent pris en argument.
    On lèvera l'exception `Wrong_rule "And Delta"` si cette formule n'est pas une conjonction.
11. Écrire une fonction `or_delta : 'a sequent -> 'a sequent` qui renvoie la prémisse de la règle $(\vdash \vee)$ appliquée à la première formule du champ `delta` du séquent pris en argument.
    On lèvera l'exception `Wrong_rule "Or Delta"` si cette formule n'est pas une disjonction.
12. Écrire une fonction `impl_delta : 'a sequent -> 'a sequent` qui renvoie la prémisse de la règle $(\vdash\rightarrow)$ appliquée à la première formule du champ `delta` du séquent pris en argument.
    On lèvera l'exception `Wrong_rule "Impl Delta"` si cette formule n'est pas une implication.
13. Écrire une fonction `not_delta : 'a sequent -> 'a sequent` qui renvoie la prémisse de la règle $(\vdash \neg)$ appliquée à la première formule du champ `delta` du séquent pris en argument.
    On lèvera l'exception `Wrong_rule "Not Delta"` si cette formule n'est pas une négation.
14. Écrire une fonction `proof_search : 'a sequent -> bool` qui renvoie `true` si le séquent pris en argument est valide, et `false` sinon, en implémentant la stratégie présentée précédemment.

Tester votre fonction sur des exemples : ceux du sujet sont fournis dans un fichier `sequent_test.ml`.

2 Réfutation par coupure

Dans cette partie, on se limite à manipuler un sous ensemble des formules du calcul propositionnel formé d'éléments qu'on appelle des clauses. Une clause est simplement une disjonction de littéraux. Les variables intervenant dans une clause C sans être précédées du connecteur $\neg$ sont appelées les variables positives de C, et les autres les variables négatives de C.

1. Rappeler l'énoncé selon lequel toute formule du calcul propositionnel est sémantiquement équivalente à une conjonction de clauses.
2. Si C est une clause dont les variables positives sont $a_1, \ldots, a_n$ et les variables négatives sont $b_1, \ldots, b_m$, montrer que C est équivalente à l'implication $(b_1 \wedge \ldots \wedge b_m) \Rightarrow (a_1 \vee \ldots \vee a_n)$.

Remarquez que le résultat précédent reste vrai, même s'il n'y a a aucune variable positive ou aucune variable négative si on considère qu'un $\wedge$ sur un ensemble vide vaut le neutre pour $\wedge$, à savoir $\top$ et que le $\vee$ sur un ensemble vide vaut le neutre pour $\vee$, à savoir $\bot$. Une clause ne contenant ni variable positive, ni variable négative est en particulier sémantiquement équivalente à $\top \Rightarrow \bot$ c'est à dire à $\bot$ et on appelera une telle clause la clause vide.

On ne considère dans la suite que des clauses simplifiées, c'est-à-dire des clauses sans littéral en doublon. Dès qu'une clause C fera intervenir un littéral en double, on commencera par supprimer les occurrences inutiles de ce doublon et on appelle cette opération la simplification de C. Dans la suite, une clause est par défaut une clause simplifiée.

La déduction par coupure est un système de déduction dont la seule règle est la règle dite de coupure. Pour toutes clauses simplifiées C_1 et C_2, on dit que la clause C se déduit par coupure de C_1 et C_2 s'il existe une variable v intervenant positivement dans C_1 et négativement dans C_2 (resp. négativement dans C_1 et positivement dans C_2) et telle que C est la simplification de la disjonction de tous les littéraux de C_1 sauf v (resp. $\neg v$) et de tous les littéraux de C_2 sauf $\neg v$ (resp. v).

On dit qu'on a coupé sur la variable v et on note :

$$\frac{C_1 \quad C_2}{C}\text{ cut}$$

Si S est un ensemble de clauses simplifiées initial, l'ensemble des clauses C prouvables par coupure à partir de S est défini inductivement par :
— Si $C \in S, C$ est prouvable à partir de S.
— Si C_1 et C_2 sont prouvables par coupure à partir de S et que C se déduit par coupure à partir de C_1 et C_2 alors C est prouvable par coupure à partir de S.

Si C est prouvable par coupure à partir de S, on note $S \vdash C$.

Exemple : $\{\neg a \vee b \vee \neg c, c \vee b, a \vee \neg d\} \vdash b \vee \neg d$. En effet, on a :

$$\dfrac{\dfrac{\neg a \vee b \vee \neg c \quad a \vee \neg d}{b \vee \neg c \vee \neg d}\, cut \qquad c \vee b}{b \vee \neg d}\, cut$$

Vu la question 2, cela ne devrait pas choquer votre intuition sémantique. En effet, $\neg a \vee b \vee \neg c \equiv a \Rightarrow (b \vee \neg c)$ et $a \vee \neg d \equiv d \Rightarrow a$. Il n'est donc pas anormal de pouvoir déduire $d \Rightarrow (b \vee \neg c) \equiv b \vee \neg d \vee \neg c$ lors de la première application de la règle de coupure dans l'arbre précédent.

Lorsque la clause vide est prouvable par coupure à partir de S, on dit que S admet une réfutation par coupure. On peut montrer que la réfutation par coupure est correcte et complète (cf DM) : autrement dit, $S \vdash \perp$ si et seulement S est non satisfiable.

2.1 IMPLÉMENTATION DE LA RÉFUTATION PAR COUPURE

L'objectif de cette partie est d'implémenter un algorithme `derive_clause_vide` dont la spécification est :

Entrée : Un ensemble de clauses (simplifiées) S.
Sortie : Oui si S admet une réfutation par coupure, non sinon.

D'après ce qui précède, un tel algorithme permet de décider si un ensemble de clauses est non satisfiable.

On propose de représenter une clause en Ocaml comme suit : une clause est une liste d'entiers relatifs non nuls. Un entier positif représente une variable positive et un entier négatif une variable négative. On demande de plus à une clause d'être une liste :
- Triée dans l'ordre croissant. Ceci sera nécessaire pour pouvoir tester l'égalité de deux clauses.
- Sans doublons, de sorte à ne considérer que des clauses simplifiées.

Par exemple, la clause $v_1 \vee v_2 \vee \neg v_4$ sera représentée par la liste [-4 ;1 ;2]. La clause vide est représentée par la liste vide.

```
type clause = int list
```

L'idée pour construire cet algorithme est le suivant : si S est un ensemble de clauses, on construit toutes les clauses prouvables par coupure à partir de S puis on vérifie si la clause vide fait partie de l'ensemble ainsi construit. Plus précisément :
- On maintient à jour un ensemble C_t de clauses à traiter et C_p de clauses prouvables. Initialement, C_t est l'ensemble de clauses initial et C_p est vide.
- Tant que C_t n'est pas vide, on en prend une clause, c. Pour tout élément $c' \in C_p$, on construit toutes les clauses déductibles par coupure de c et c'. Toutes les clauses ainsi construites qui ne sont ni déjà dans C_p, ni déjà dans C_t sont rajoutées à C_t. Une fois qu'on a généré toutes les clauses possibles à partir de c et des clauses de C_p, on ajoute c à C_p.
- Lorsque C_t est vide, on vérifie si la clause vide est dans C_p.

1. Ecrire une fonction coupure de signature `clause -> clause ->int -> clause`. Elle prend en entrée deux clauses c_1, c_2 et un entier n qu'on supposera apparaitre positivement dans c_1 et négativement dans c_2. Elle renvoie une liste triée sans doublon correspondant à la clause obtenue par coupure à partir de c_1 et c_2 sur la variable n.
2. Ecrire une fonction `variables_a_couper` de signature `clause -> clause -> int list` telle que `variable_a_couper c1 c2` renvoie la liste des variables v de c_1 telles qu'on puisse appliquer la règle de coupure à c_1 et c_2 en coupant sur v.

3. Déduire des questions précédentes une fonction `nouvelles_clauses` de signature `clause -> clause -> clause list` telle que `nouvelles_clauses c1 c2` renvoie la liste des clauses déductibles par coupure à partir de c_1 et c_2.
4. Ecrire une fonction `exists_clause_vide` de signature `clause list -> bool` indiquant si la clause vide fait partie de la liste de clauses en entrée.
5. Déduire de tout ce qui précède une fonction `derive_clause_vide` de signature `clause list-> bool` telle que `derive_clause_vide lc` renvoie `true` si la clause vide est prouvable par coupure à partir de l'ensemble de clauses `lc` et `false` sinon.

On pourra s'aider de fonctions auxilliaires et des fonctions du module `List`.

La fonction précédente est excessivement naïve. Une amélioration immédiate serait d'ailleurs de cesser de construire des clauses prouvables dès que la clause vide a été obtenue. La fin de cette partie est un bonus dans lequel on explore deux stratégies visant à limiter la complexité en pratique de `derive_clause_vide`.

7. Jusqu'à présent, les clauses $c \in C_t$ avec lesquelles on tente successivement de créer de nouvelles clauses prouvables sont choisies au hasard. Comment pourrait-on choisir c pour espérer obtenir la clause vide rapidement ? Proposer une heuristique de choix de c et l'implémenter.
8. Une clause simplifiée est dite tautologique si elle fait intervenir une variable à la fois positivement et négativement.
   (a) Montrer que, si $S \vdash \emptyset$ alors la réfutation par coupure de $\emptyset$ à partir de S peut se faire sans utiliser de clause tautologique. On en déduit qu'ajouter les clauses tautologiques à C_t est inutile à la correction de la réfutation par coupure.
   (b) En déduire que pour chaque $c \in C_t$ et chaque $c' \in C_p$, ne construire qu'une seule des clauses déductibles à partir de c et c' (plutôt que toutes) est suffisant pour conserver la correction de `derive_clause_vide`.
   (c) Modifier `derive_clause_vide` de façon à ignorer les clauses tautologiques.

2.2 Le retour du club écossais

L'objectif de cette partie est de faire prouver un résultat sémantique à votre machine à l'aide de derive_clause_vide (sous réserve que cette fonction soit correcte, résultat que l'on délègue à la partie 3).

Il existe en Ecosse un club très fermé dont les membres obéissent aux règles suivantes :

(1) Tout membre non écossais porte des chausettes rouges.
(2) Tout membre portant des chaussettes rouges porte un kilt.
(3) Les membres mariés ne sortent pas le dimanche.
(4) Un membre sort le dimanche si et seulement si il est écossais.
(5) Tout membre qui porte un kilt est écossais et est marié.
(6) Tout membre écossais porte un kilt.

9. Modéliser les règles du club en logique propositionnelle. En déduire une formule caractérisant le fait de pouvoir entrer dans le club écossais.
10. En déduire un ensemble de clauses S tel que S est satisfiable si et seulement si il est possible d'entrer dans le club. En utilisant la partie 1 et sans dresser de table de vérité à 32 lignes (!), monter que personne ne peut entrer dans le club.