TP Option informatique $n°1$ – Calcul des séquents

Merci à Anthony Lick dont un sujet a été essentiellement repris ici.

Dans tout le sujet, on se donne un ensemble dénombrable $\mathcal{V}$ de variables. Et on s'intéresse aux formules propositionnelles que l'on peut former sur cet ensemble de variables.

On a vu en cours la déduction naturelle : c'est un système de preuve correct et complet, c'est-à-dire que les formules prouvables pour la déduction naturelle sont exactement les tautologies. On a cherché ensemble des arbres de preues : c'était parfois pénible. En effet, il n'existe pas de stratégie de preuve générale pour chercher un arbre de preuve.

On présente ici un autre système de preuve : le *calcul des séquents*. Un séquent de la déduction naturelle est de la forme $\Gamma \vdash f$: 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 générale de séquent : un séquent du calcul des séquents est de la forme $\Gamma \vdash \Delta$, où Γ *et* Δ sont des ensembles finis de formules. Précisément, l'ensemble des séquents du calcul des séquents est défini inductivement à l'aide des règles présentées ci-dessous.

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

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

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

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

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

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

Un séquent du calcul des séquents $\Gamma \vdash \Delta$ doit se comprendre de la façon suivante : "à partir de la **conjonction** des formules de Γ, on peut démontrer **la disjonction** des formules de Δ", et le cas des séquents de la déduction naturelle où Δ ne comporte qu'une seule formule est alors un simple cas particulier. Le séquent $f_1, \ldots, f_n \vdash g_1, \ldots, g_n$ se lit donc : "avec les hypothèses $f_1, \ldots, f_n$, je peux démontrer l'une des formules $g_1, \ldots, g_n$".

Caractère moral des règles

1. Expliquer brièvement pourquoi les règles précédentes sont raisonnables vis-à-vis de la sémantique proposée ci-dessus.

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 G.

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 au dessus) 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.

Quelques exemples

On commence par prouver quelques règles simples à l'aide de ce nouveau système de preuve. Bien que sémantiquement très claires, ces règles sont parfois difficiles à obtenir avec la déduction naturelle (certaines ont été traitées en cours, mais vous pouvez essayer de montrer les autres et comparer les difficultés rencontrées avec l'un ou l'autre formalisme).

2. Établir le tiers exclus $\dfrac{}{\Gamma \vdash f \vee \neg f}$te.
3. Établir la loi de De Morgan suivante : $\dfrac{}{\neg(p \vee q) \vdash \neg p \wedge \neg q}$.
4. Établir la loi de De Morgan suivante : $\dfrac{}{\neg p \wedge \neg q \vdash \neg(p \vee q)}$.

Correction

5. En mettant en forme les arguments donnés dans la question 1, montrer que le calcul des séquents est un système de preuve *correct*, c'est-à-direque pour toute formule f et tout ensemble de formules Γ, si on a $\Gamma \vdash f$ alors on a $\Gamma \models f$.

   On raisonnera pas induction structurelle.

On peut également montrer que le calcul des séquents est un système de preuve complet, c'est-à-dire que pour toute formule f et tout ensemble de formules Γ, si on a $\Gamma \vdash f$ alors on a $\Gamma \models f$.

Implémentation et algorithme de recherche de preuve

On va comme d'habitude prendre le type `char` pour les variables propositionnelles. On se donne le type `prop` suivant :

```
type prop =
  | Top (* formule vraie *)
  | Bot (* formule fausse *)
  | V of char (* variable *)
  | Not of prop
  | And of prop * prop
  | Or of prop * prop
  | Impl of prop * prop;;
```

Dans la suite, il sera pratique de séparer Γ (respectivement Δ) en deux parties : une ne contenant que des variables, $\top$ ou $\bot$; et une autre pouvant également contenir les autres formules de Γ (respectivement Δ). On se donne donc le type `sequent` suivant :

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

Lorsqu'on initialisera un séquent, on mettra toutes les formules dans les listes `gamma` et `delta`, y compris celles qui sont seulement composées d'une variable, $\top$ ou $\bot$, mais au cours du traitement des séquents, on pourra être amenés à déplacer des formules de ce type (et seulement de ce type) de `gamma` dans `gamma_var` ou de `delta` dans `delta_var`.

Dans la suite, on pourra utiliser sans scrupules la fonction `List.mem`.

6. Écrire une fonction `create_sequent : prop list -> prop list -> sequent` telle que `create_sequent l_gamma l_delta` retourne un `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.

7. Écrire une fonction `bot : sequent` $\to$ `bool` qui teste si la règle ($\bot$) peut être appliquée au séquent pris en argument.

   Attention : on ne suppose pas que le séquent vient d'être créé : il est possible qu'une formule $\bot$ ait été déplacée dans `gamma_var`, faut donc chercher dans `gamma` **et** dans `gamma_var`.

8. Écrire une fonction `top : sequent` $\to$ `bool` qui teste si la règle ($\top$) peut être appliquée au séquent pris en argument.

   Attention : chercher dans `gamma` **et** dans `gamma_var`.

9. Écrire une fonction `axiom : sequent` $\to$ `bool` qui teste si la règle (Ax) est applicable au séquent pris en argument.

   Attention : la formule à trouver peut être dans `gamma` **ou** dans `gamma_var`, et dans `delta` **ou** `delta_var`.

Pour la suite, la stratégie de preuve proposée ici est très simple :

- Si on peut appliquer (Ax) ou ($\top$) ou ($\bot$), on l'applique et on a prouvé le séquent.
- Sinon :
  - $\rightsquigarrow$ si `gamma` n'est pas vide, on regarde la première formule de la liste :
    - si c'est une variable, $\top$ ou $\bot$, 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 ;
  - $\rightsquigarrow$ si `gamma` est vide mais pas `delta` on procède de manière similaire avec `delta`.
  - $\rightsquigarrow$ si `gamma` et `delta` sont vides, et que les règles (Ax), ($\top$) et ($\bot$) ne s'appliquent pas, alors aucune règle ne s'applique, et le séquent n'est pas valide.

On commence donc par implémenter chacune des règles par une fonction OCaml.

En accord avec la stratégie présentée ci-dessus, si la première formule de `gamma` (respectivement `delta`) n'est pas celle sur laquelle on peut appliquer la règle considérée, on provoquera l'exception `Failure message` où `message` est une chaîne de caractère adaptée, à l'aide de la syntaxe `failwith message`.

10. Écrire une fonction `and_gamma : sequent` $\to$ `sequent` qui retourne la prémisse de la règle ($\wedge \vdash$) appliquée à la première formule du champ `gamma` du séquent donné en argument.

    On porovoquera l'exception `Failure "And Gamma"` si cette formule n'est pas une conjonction.

11. Écrire une fonction `or_gamma : sequent` $\to$ `sequent * sequent` qui retourne les prémisses de la règle ($\vee \vdash$) appliquée à la première formule du champ `gamma` du séquent donné en argument.

    On porovoquera l'exception `Failure "Or Gamma"` si cette formule n'est pas une disjonction.

12. Écrire une fonction `impl_gamma : sequent` $\to$ `sequent * sequent` qui retourne les prémisses de la règle ($\to \vdash$) appliquée à la première formule du champ gamma du séquent donné en argument.

    On porovoquera l'exception `Failure "Impl Gamma"` si cette formule n'est pas une implication.

13. Écrire une fonction `not_gamma : sequent` $\to$ `sequent` qui retourne la prémisse de la règle ($\neg \vdash$) appliquée à la première formule du champ `gamma` du séquent donné en argument.

    On porovoquera l'exception `Failure "Not Gamma"` si cette formule n'est pas une négation.

14. Écrire une fonction `and_delta : sequent -> sequent * sequent` qui retourne les prémisses de la règle ($\vdash \wedge$) appliquée à la première formule du champ `delta` du séquent donné en argument.

    On porovoquera l'exception `Failure "And Delta"` si cette formule n'est pas une conjonction.

15. Écrire une fonction `or_delta : sequent → sequent` qui retourne la prémisse de la règle ($\vdash \vee$) appliquée à la première formule du champ `delta` du séquent donné en argument.

    On porovoquera l'exception `Failure "Or Delta"` si cette formule n'est pas une disjonction.

16. Écrire une fonction `impl_delta : sequent → sequent` qui retourne la prémisse de la règle ($\vdash\rightarrow$) appliquée à la première formule du champ `delta` du séquent donné en argument.

    On porovoquera l'exception `Failure "Impl Delta"` si cette formule n'est pas une implication.

17. Écrire une fonction `not_delta : sequent → sequent` qui retourne la prémisse de la règle ($\vdash \neg$) appliquée à la première formule du champ `delta` du séquent donné en argument.

    On porovoquera l'exception `Failure "Not Delta"` si cette formule n'est pas une négation.

Ouf.

18. Écrire une fonction `is_ok : sequent -> bool` qui retourne `true` si le séquent pris en argument est valide, et `false` sinon, en implémentant la stratégie présentée précédemment.

Tester votre fonction sur les exemples suivants (copier-coller) :

```
(* Exemples renvoyant false *)
is_ok (create_sequent [] [Or(V 'a', V 'a')]);;
is_ok (create_sequent [] [Impl(Impl(Impl(V 'a', V 'b'),V 'a'),V 'b')]);;

(* Exemples renvoyant true *)
is_ok (create_sequent [] [Or(V 'a', Not(V 'a'))]);;
is_ok (create_sequent [] [Impl(Impl(Impl(V 'a', V 'b'),V 'a'),V 'a')]);;
is_ok (create_sequent [And(And(V 'a', V 'b'), V 'c')] [And(V 'a', And(V 'b', V 'c'))]);;
is_ok (create_sequent [Or(Or(V 'a', V 'b'), V 'c')] [Or(V 'a', Or(V 'b', V 'c'))]);;
is_ok (create_sequent [Impl(V 'a', V 'b')] [Impl(Not(V 'b'), Not(V 'a'))]);;
is_ok (create_sequent [Impl(Not(V 'b'), Not(V 'a'))] [Impl(V 'a', V 'b')]);;
is_ok (create_sequent [V 'a'; V 'b'] [Impl(V 'c', And(V 'a', V 'c'))]);;
```

Il est possible aussi d'en tester d'autres !