Programmation Avancée

Devoir à la maison

# Prolog

David Baelde

Ce devoir à la maison tient lieu de partiel. Il est à rendre pour le 7 mai. Une base de code OCaml est disponible en ligne, contenant notamment des signatures et des tests.

Vous veillerez à respecter la structure imposée par les fichiers en place dans la base de code. Vous pouvez modifier les tests (mieux, en ajouter) mais lors de la correction j'écraserai les fichiers de test par les originaux.

J'attends un retour par mail, avant le TP du 7 mai, sous la forme d'un tarball contenant le code et le Makefile. Certaines questions demandent des explications mais celles-ci peuvent être données sous forme de commentaires dans le code. Sinon, vous pouvez inclure un simple fichier `README`.

## 1 Introduction

Nous allons réaliser en OCaml un prototype d'implémentation du langage Prolog. Nous en rappelons les bases : un *programme* est donné par un ensemble de clauses de Horn en logique du premier ordre ; l'exécution du programme consiste en la recherche de solutions pour une *requête* consistant en une formule logique simple (sans implication ni négation) contenant potentiellement des variables libres, interprétées existentiellement, et appelées *variables logiques*.

Par exemple, le programme suivant spécifie l'addition :

```
plus 0 X X.
plus (s X) Y (s Z) :- plus X Y Z.
```

Ce programme étant chargé, la requête `plus (s 0) (s 0) X` mène à l'unique solution `X := s (s 0)` tandis que `plus (s (s 0)) X (s (s (s 0)))` mène à `X := s 0`. Enfin, la requête `plus 0 X X` produit l'unique solution `X := X`.

La stratégie de recherche de preuve de Prolog est extrêmement simple. Pour l'expliquer (et l'implémenter) il est pratique d'étendre légèrement le langage des clauses de Horn en ajoutant le prédicat d'égalité et la disjonction.

Pour résoudre une requête conjonctive, on cherche à résoudre l'une après l'autre les deux sous-formules, en propageant les instantiations de variables

logiques. Pour une requête disjonctive, on cherche à résoudre l'une des sous-formules, dans l'ordre. Ainsi, la requête `(X = s 0 \/ X = 0) /\ plus X 0 0` va mener à la tentative de résolution de `plus (s 0) 0 0`, qui échoue, puis à `plus 0 0 0` qui réussit, d'où l'unique solution `X := 0`.

Pour trouver une solution à une requête atomique de la forme `p t` on cherche une clause du programme dont la tête s'unifie avec `p t` puis on cherche à résoudre comme sous-requête le corps de la clause. Ceci est fait selon un parcours en profondeur, en prenant les clauses dans l'ordre de leur déclaration. Ainsi, la requête `plus X 0 X` mène à l'énumération des solutions `X := 0`, `X := s 0`, `X := s (s 0)`, etc. Si l'on inversait l'ordre des clauses, cette dernière requête provoquerait une exécution infinie avant même de produire une seule solution. Cette simplicité peut paraître excessive du point de vue de la preuve automatique, mais elle est justifiée si l'on voit Prolog comme un langage de programmation, dont la sémantique doit être prévisible pour le programmeur.

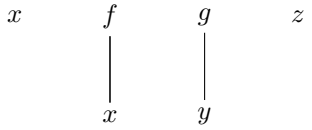
# 2 Représentation des termes

Dans cette première partie, nous implémentons les modules `Term` et `Unify`. Le choix de la structure de données représentant les termes doit prendre en compte l'utilisation intensive de l'unification et du backtracking dans un interprète Prolog. La solution proposée permet de combiner les avantages d'une unification destructive (instantiation en place) et de la programmation persistante (ne pas avoir à copier des termes en prévision du backtracking).

## 2.1 Représentation interne

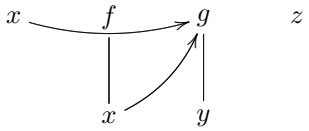
La représentation des termes doit permettre l'instantiation en place des variables. Pour cela on représente chaque variable par une référence. Cette référence sera partagée entre les occurrences d'une variable, de sorte que la modification de la référence permette de modifier instantanément toutes les occurrences. Il faudra de plus veiller à ce que le partage soit maintenu quand on effectue une instantiation.

On représente graphiquement un exemple du comportement désiré, où l'on considère les termes $x$, $f(x)$, $g(y)$ et $z$ :

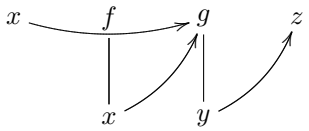



Ici, les deux occurrences de $x$ sont représentées par la même référence. Si l'on souhaite substituer $x$ par $f(y)$ on modifie cette référence, créant ainsi la situation

suivante :




Le premier terme dénotait initialement la variable $x$. Si on le lit maintenant, comme la variable est instantiée on va suivre le lien, et obtenir $g(y)$ — du point de vue des termes représentés la variable $x$ n'existe plus mais est devenue une redirection. Si l'on substitue maintenant $y$ par $z$ on obtient :




Le second terme, anciennement $f(x)$, représente maintenant $f(g(z))$. Si au lieu de substituer $y$ par $z$ on avait voulu substituer $z$ par le premier terme (celui qui était initialement $x$) on aurait eu deux possibilités : on peut créer un lien de $z$ vers $x$, ou bien directement créer un lien de $z$ vers le noeud $g$ pointé par $x$. La seconde possibilité est bien sûr préférable puisqu'elle permet de limiter les successions d'indirections. On pourra noter ici une forte similitude avec la structure de données *union-find*.

**Question.** Commencez l'implémentation du module `Term` : définissez un type `in_term` et une fonction `bind` qui permettent les opérations décrites ci-dessus. On s'aidera d'une fonction auxiliaire `deref` qui suit les redirections créés par les instantiations afin d'accéder au contenu effectif d'un `in_term`.

**Question.** On définira ensuite la fonction `observe` qui renvoie un `term` décrivant le contenu effectif d'un `in_term`. L'idée de la signature du module `Term` est que l'utilisateur du module ne doit pas avoir à se préoccuper de la représentation interne des termes, des redirections, etc. Ainsi, quand `observe` renvoie une variable il doit bien s'agir d'une variable qui n'a pas encore été instantiée ; sinon c'est le contenu de son instantiation qui devra être observé.

**Question.** On implémentera enfin les fonctions de test d'égalité pour les variables et les termes (`equals` et `var_equals`) ainsi que les constructeurs de termes `make`, `fresh` et `var`, documentées comme les autres dans le fichier `mli`.

De façon optionnelle on pourra ajouter à la signature de `Term` une fonction pour afficher des termes. Cela n'est pas demandé mais un afficheur (même très basique) peut être utile pour déboguer le module lui-même mais aussi les composants que nous allons développer ensuite.

### 2.2 Annulation efficace des instantiations

Notre interprète va faire une recherche avec backtracking, et il ne faut pas que les instantiations de variables faites dans une branche de la recherche persistent dans une branche alternative explorée ensuite. Il serait trop inefficace d'éviter cela en copiant l'état des termes. Au lieu de cela nous allons maintenir un historique des instantiations effectuées, afin de pouvoir les annuler pour retourner à un état précédent.

**Question.** Mettez en place une structure de données appropriée pour maintenir cet historique, dans laquelle vous stockerez à chaque appel de `bind` suffisamment d'information pour pouvoir défaire l'instantiation. Définir le type `state` ainsi que les opérations `save` et `restore`.

Vous devriez maintenant pouvoir compiler et exécuter `term_test`, qui effectue deux tests basiques.

### 2.3 Unification

Maintenant qu'on a implémenté nos termes et caché les complexités de cette implémentation derrière une interface suffisamment abstraite, il va être facile d'implémenter l'algorithme d'unification.

Étant donné notre représentation des termes, notre algorithme d'unification va adopter un style "destructif" : il va modifier en place ses données en appliquant l'unificateur. Par exemple, `unify x t` ne renverra rien mais aura pour effet d'instantier `x` par `t`.

Comme dans la plupart des implémentations de Prolog on n'implémentera pas la vérification d'occurrence (occurs-check) car cela entrainerait un surcout assez gros (qui masquerait notamment les optimisations que nous serons amenés à faire dans la suite). Du coup, des termes circulaires pourraient être crées par l'unification. Il faudra s'en méfier, mais on ne demande pas que le code prenne des dispositions pour les supporter.

**Question.** Implémenter `Unify.unify`, dont la signature est encore une fois donnée. Si les termes sont unifiables l'unificateur le plus général est directement appliqué aux variables. Sinon, l'exception `Unification_failure` doit être levée.

Vérifiez votre implémentation à l'aide du test `unify_test` fourni dans la base de code.

## 3 Le langage Prolog

Dans cette partie nous définissons les objets manipulés par l'interprète Prolog : buts et programmes logiques. Afin de se concentrer sur l'essentiel notre interprète Prolog ne comportera par d'interface textuelle (lecture du programme

logique depuis un fichier) mais on construira les programmes directement dans le code OCaml.

Votre tâche est d'implémenter le module `Prolog`, dont la signature est donnée. Le type des buts est donné explicitement par cette signature ; il décrit de façon évidente des formules simples de la logique du premier ordre. Le type des programmes est abstrait, il vous faudra le définir, avec les opérations demandées par la signature.

Les programmes logiques sont construits par concaténation (`concat_prog`) à partir du programme vide (`prog_empty`) ou bien d'un programme correspondant à un prédicat, fabriqué par l'opération `make_predicate`. Ce dernier prend en argument le nom du prédicat dont on construit le programme, ainsi que la liste des clauses définissant ce prédicat. Ces clauses sont elles-mêmes construites à l'aide des constructeurs `horn` et `forall`. Par exemple, le programme logique de l'addition, donné au début de ce sujet, s'écrira comme suit :

```
let z = Term.make "z" []
let s x = Term.make "s" [x]
let plus n m p = Atom ("plus",[n;m;p])
let (<==) hd body = horn hd body
let prog =
  make_predicate "plus"
    [ forall (fun x ->
        plus z x x <== tt) ;
      forall (fun x ->
      forall (fun y ->
      forall (fun z ->
        plus (s x) y (s z) <== plus x y z))) ]
```

L'opération `forall` utilise la notion de liaison d'OCaml (`fun x ->` ...) pour simuler le lieur ($\forall x.\ \ldots$) de la logique du premier ordre, selon le principe de la *syntaxe abstraite d'ordre supérieur*. On pourrait imaginer [1], par exemple, que son implémentation soit la suivante, instantiant immédiatemment la variable quantifiée par une variable fraîche :

```
let forall f = f (Term.var (Term.fresh ()))
```

Outre les constructeurs de programmes, la seule fonctionalité demandée est la fonction `lookup` : étant donné un nom de prédicat `p` et une liste de termes `l` représentant les paramètres de ce prédicat, `lookup program p l` doit renvoyer le but que l'interprète Prolog doit résoudre pour prouver `p l`. Par exemple, `lookup program "plus"[t1;t2;t3]` devrait donner le but

$$\begin{aligned}&(t_1 = 0 \wedge t_2 = X_1 \wedge t_3 = X_1 \wedge \top)\\ \vee\ &(t_1 = s(X_2) \wedge t_2 = Y_2 \wedge t_3 = s(Z_2) \wedge \text{plus } X_2\ Y_2\ Z_2)\end{aligned}$$

1. Attention, cette implémentation de `forall` n'est probablement pas le meilleur point de départ pour une implémentation du module `Prolog`. En fait, on veut éviter d'avoir à "rafraîchir" les variables libres d'une clause pour en obtenir une nouvelle instance. C'est possible si on appelle la fonction de construction passée à `forall` à chaque fois qu'on veut créer une instance fraîche de la clause.

où les variables $X_1$, $X_2$, $Y_2$ et $Z_2$ sont fraîches et correspondent aux variables libres (moralement quantifiées universellement) dans les clauses du programme du prédicat plus.

**Question.** Implémenter le module `Prolog` en définissant un type `program`, la fonction `lookup` et les fonctions de construction.

De façon optionnelle, on pourra définir un `prog_empty` pas vraiment vide mais qui définisse un prédicat unaire `print` qui s'évalue toujours en $\top$ mais affiche par effet de bord son argument. Ce prédicat pourra être utile pour déboguer les interprètes Prolog.

# 4 Interprète en CPS

Nous allons enfin pouvoir implémenter un interprète Prolog. Nous commençons par une implémentation en CPS, selon le style vu en TP avec les continuations de succès et d'échec.

En Prolog, le succès est une instantiation des variables libres de la requête. Vu notre façon de manipuler les termes, cette substitution n'a pas de représentation explicite : elle a simplement été appliquée en mémoire. Ainsi le succès ne sera pas retourné explicitement mais sera l'état de la mémoire au moment où la continuation de succès est appelée. Le type de la continuation d'échec sera donc `'a fk = unit -> 'a` tandis que la continuation de succès aura pour type `'a sk = 'a fk -> 'a`.

Il faudra de plus veiller à ce que les succès (instantiations) soient bien "défaits" après avoir été signalés à la continuation d'échec. On recommande de traiter dans le même cas (celui de l'égalité) toutes les opérations modifiant l'état des variables (unification, sauvegarde et restauration de l'état).

**Question.** Implémenter le module `Cps` : on écrira une fonction
`val search : program -> goal -> 'a sk -> 'a fk -> 'a`
qui énumère toutes les solutions d'une requête, puis on la spécialisera en
`val has_solution : program -> goal -> bool`
Cette dernière fonction vous permettra de compiler et d'exécuter `prolog_test`.

On veillera bien à ce que le code soit en CPS. En particulier, tout appel récursif doit être terminal. Je vous recommande fortement de tester si c'est le cas en exécutant une requête qui boucle : si vous obtenez une `Stack_overflow` (ou une `segfault` dans le cas d'un programme compilé en natif) c'est qu'il y a un soucis.

# 5 Vers une machine abstraite

L'implémentation de l'interprète précédent est naturelle pour qui maîtrise la programmation en CPS. Cependant elle requiert un langage hôte d'ordre supérieur (ici, OCaml) qui ne serait pas un choix naturel pour l'implémentation de l'environnement d'exécution d'un Prolog réaliste, qui s'appuierait plutôt sur une machine abstraite implémentée en C, permettant un contrôle plus précis de la mémoire.

Dans cette partie nous allons faire évoluer notre programme vers une telle machine abstraite. Nous resterons au sein du langage OCaml, et continuerons notamment d'utiliser son gestionnaire de mémoire, mais cette étape devrait déja nous permettre d'observer un gain de performance, et surtout de découvrir la structure générale des machines abstraites pour Prolog.

**Question.** Défonctionaliser la procédure `Cps.search`. Le résultat sera mis dans le module `Defunc`, qui devra offrir une fonction `has_solution` de même type que précédemment. En décommentant les dernières lignes dans `prolog_test.ml` vous pourrez tester que ce second interprète se comporte bien comme le premier.

Dans cette première étape il vous est demandé de faire la défonctionalisation de la façon la plus systématique possible, en introduisant autant de nouveaux types de données que nécessaire, et de façon à ce qu'on voie bien la correspondance avec l'interprète en CPS.

**Question.** Implémentez dans `Abstract_machine` un interprète aussi simplifié que possible, en optimisant éventuellement certaines constructions. On pourra notamment se demander si certains enchaînements de constructions sur la continuation d'échec ne peuvent pas se simplifier en une seule. On commentera le code en expliquant brièvement la sémantique des différents objets manipulés par la machine abstraite. On testera de nouveau avec `prolog_test`.

# 6 Bonus : coupures

La programmation logique telle que définie jusqu'ici est dite *pure*. En effet, les programmes sont des formules logiques en bonne et due forme dont le sens est complètement déterminé par la logique, à l'exception de problèmes de non-terminaison liés à la stratégie de recherche de preuve adoptée.

Bien souvent, la programmation logique pure est trop contraignante pour obtenir des algorithmes aussi efficaces qu'on le souhaite. De ce fait, des opérateurs *impurs* ont été ajoutés aux langages de programmation logique. Considérons par exemple le programme suivant, où `le` est supposé représenter la relation d'ordre non-strict sur les entiers naturels :

```
max X Y X :- le Y X.
max X Y Y :- le X Y.
```

Logiquement, ce programme est correct et rien n'y est superflu. Si l'on souhaite utiliser ce programme pour calculer le maximum de deux entiers $n$ et $m$ on observe cependant des calculs inutiles :

- L'interprète Prolog va d'abord vérifier $m \leq n$. Si c'est le cas, il va proposer une première réponse : le maximum est $n$. En cas de backtracking, il va tenter ensuite d'obtenir une solution par le biais de la seconde clause, mais la condition $n \leq m$ ne peut réussir que si $m = n$, ce qui mène à une seconde solution identique à la première.
- Si $m \leq n$ échoue, l'interprète tente d'utiliser la seconde clause et va alors vérifier $n \leq m$. Mais comme $n$ et $m$ sont des termes clos décrivant des entiers naturels, ce test est inutile car on sait déja $\neg(m \leq n)$, c'est à dire $n < m$.

L'opérateur de coupe-choix (en anglais, *cut*) a été introduit pour éviter ces lourdeurs. Il s'agit d'une variable propositionnelle qui est logiquement vraie mais a un comportement opérationnel spécifique : quand elle est rencontrée, Prolog abandonne toutes les alternatives de preuve ouvertes depuis le dernier but atomique rencontré. En d'autres termes, le coupe-choix réalise un *commit* sur la façon dont le prédicat courant peut être prouvé.

On réécrit l'exemple précédent avec un coupe-choix, noté ! :

```
max X Y X :- le Y X, !.
max X Y Y.
```

Ce programme est correct, mais la logique ne suffit plus à l'expliquer : si ! est lu comme $\top$, alors on devrait avoir `max 3 2 3` mais aussi `max 3 2 2` par la deuxième clause. Pour comprendre ce qu'il se passe il faut s'appuyer sur le comportement opérationnel précis de l'interprète. Quand il rencontre le but `max n m Z`, celui-ci va d'abord essayer la première clause, et vérifier `le m n`. Si c'est le cas, il atteint le but !, ce qui coupe (supprime) l'alternative encore ouverte consistant à utiliser la deuxième clause. Si ce n'est pas le cas, alors il va essayer la seconde clause, qui ne nécessite pas de vérification d'ordre puisqu'on sait déja que $n < m$.

**Question.** Vous trouverez dans le fichier `bonus.ml` un programme Prolog qui manipule des arbres rouges et noirs. Un test est exécuté dans ce fichier : il construit un arbre $t$ en insérant successivement les entiers de 1 à 500 dans un arbre initialement vide, puis vérifie que l'entier 0 n'est pas dans $t$.

1. Expliquez pourquoi ce test est inutilement coûteux.
2. Étendez l'un de vos interprètes Prolog [2] pour supporter le coupe-choix.
3. Utilisez le coupe-choix pour optimiser le programme logique du test. Un gain de 30% devrait être possible. Pour cette question je m'attends à ce que vous modifiez le fichier `bonus.ml`, et je n'écraserai pas vos modifications.

2. Il est probablement plus simple de partir de la machine abstraite optimisée, mais cela dépend de ce que vous aurez fait exactement et une solution en CPS serait intéressante et faisable aussi.