TP8 – Modules, algorithme d'unification de Herbrand

Alexandre Drewery

Dans ce prélude au projet de programmation de cette semaine, nous allons créer un petit système de gestion et d'unification de termes.

1 Travailler avec plusieurs fichiers

1.1 Modules

Coder en Ocaml pendant trois jours remplira certainement plus d'un fichier et ce TP est l'occasion de rappeler comment le multi-fichier fonctionne en Ocaml.
Un **module** est un espace nommé qui contient des définitions (de fonctions, de types, de valeurs, etc.). Le nom d'un module commence toujours par une majuscule. Vous avez déjà utilisé des modules, par exemple le module `List` ou le module `Array`. En Ocaml, un fichier `fichier.ml` peut être compilé en un module `Fichier`.
En guise d'exemple voici deux fichiers `a.ml` et `b.ml` :

```
let stra = "je suis dans le module A !\n"

let say_thing : unit -> unit = fun () -> print_string stra
```

```
let strb = "je suis dans le module B !\n"

let () = print_string A.stra
let () = print_string strb

let () =
  let open A in
  print_string stra
```

Les définitions du module `A` peuvent être appelées depuis le module `B` en les préfixant d'un `A.`. Vous pouvez aussi ouvrir le module `A` à l'aide de `open` (dans un let binding comme dans l'exemple ou au top level).
Vous pouvez compiler ces deux fichiers avec la commande suivante (l'ordre de `a.ml` et `b.ml` est important puisque `B` utilise `A`) :

```
ocamlc -o b.exe a.ml b.ml
```

1.2 Interfaces

Si un module a des définitions qui ne servent qu'à son fonctionnement interne et ne devraient pas être manipulées par un utilisateur extérieur, vous pouvez opacifier votre module à l'aide d'une interface. En Ocaml, l'interface d'un fichier `fichier.ml` est représentée dans un fichier `fichier.mli`. C'est l'analogue du `.h` en C.
On pourrait imaginer l'interface suivante pour le module `A` dans un fichier `a.mli` (un exemple plus complet de `.mli` est donné dans l'exercice d'aujourd'hui) :

```
val say_thing : unit -> unit
```

Cette interface cache complètement la valeur `stra`, qui n'est plus accessible depuis l'extérieur. Ainsi le module `B` ne peut plus réaliser ses `print_string A.stra`. Pour compiler avec une interface vous pouvez l'ajouter avant le `.ml` associé dans la ligne de compilation :

```
ocamlc -o b.exe a.mli a.ml b.ml
```

Dans une interface vous pouvez :
- définir des valeurs/fonctions à l'aide du mot-clé `val`. Vous devez donner leur type et c'est un bon endroit pour donner la documentation.
- définir des types "concrets" par exemple `type 'a tree = L of 'a | N of 'a * 'a tree * 'a tree`. Les constructeurs et la structure d'un type concret sont visibles depuis l'extérieur ce qui permet à un utilisateur de déconstruire les éléments du type par exemple.
- définir des types "abstrait", par exemple `type 'a tree`. L'utilisateur a connaissance du type mais pas de sa structure interne et ne peut interagir avec que par le biais des fonctions que vous lui fournissez.
- définir d'autres choses dont nous parlerons une autre fois.

2 Unification de termes

Le système de gestion de termes du projet de programmation est une bonne occasion de travailler avec les notions de modules et d'interface.
Pour définir ce que sont les termes, on suppose qu'on s'est donné un ensemble de *variables* et un ensemble de *symboles de fonctions*. L'ensemble des symboles de fonction est muni d'une fonction d'arité : l'arité d'un symbole de fonction indique le nombre d'arguments attendus.
Les **termes** sont des arbres de syntaxe abstraits construits à partir des variables en utilisant les symboles de fonction. Autrement dit, l'ensemble des termes est défini inductivement comme suit :
- les variables sont des termes ;
- si $t_1, \ldots, t_k$ sont des termes et que f est un symbole de fonction d'arité k alors $f(t_1, \ldots, t_k)$ est un terme.

Quelques exemples de termes définis à partir des symboles de fonction z (arité zéro) et s (arité un), et des variables X et Y :

```
z
s(X)
s(z)
Y
s(s(Y))
```

Les variables sont naturellement faites pour être **instanciées** (remplacées) par des termes plus complexes. Le fichier `term.mli` contient l'interface d'un module de gestion de termes et de manipulation d'une instanciation des variables.

1°) Proposez une implémentation `term.ml` du module `Term`. Aidez-vous de la documentation de l'interface et des indications et conseils suivants :
- le module contient une instanciation courante des variables, qui peut être exportée via le type `state`. Vous pouvez par exemple utiliser une Map.
- il y a deux types qui représentent les termes : un type `t` qui est abstrait (impossible pour un utilisateur extérieur de déconstruire un élément du type `t`) et le type `obs_t` qui correspond aux termes *observés* à la racine à l'aide de la fonction `observe` en se basant sur l'instanciation actuelle des variables. Je vous conseille de poser `type t = obs_t` pour votre première implémentation.
- si une variable x est actuellement instanciée en un terme t, alors son observation doit rendre le terme t.
- vous avez le choix du type abstrait `var` mais rappelez-vous que vous devez être capable de générer des variables *fraiches* (neuves, encore jamais vues auparavant).
- la fonction d'égalité entre termes doit tenir compte de l'instanciation des variables (et donc faire appel à `observe`).

2°) Proposez une implémentation `unify.ml` du module `Unify`. La fonction `unify` doit implémenter l'algorithme vu en classe.
Vous pouvez compiler et exécuter les fichiers fournis pour tester vos implémentations.