TP 9: Unification

Ioana Pasca, Marc Lasson

1 Rappels de Caml

1.1 Référence et égalité physique

En `Objective Caml`, les références se manipulent à l'aide des trois constructions suivantes :

- `ref t` : crée une nouvelle référence initialisée avec la valeur de `t`.
- `r := t` : remplace le contenu de la référence `r` par la valeur de `t`.
- `!r` : retourne le contenu de la référence `r`.

Il est important de comprendre que les références sont des valeurs à part entière et qu'elles peuvent être prises en paramètre ou retournées par des fonctions. Une bonne façon de vérifier que vous avez compris serait de comprendre le programme suivant.

```
let r1 = ref (17*43) in
let r2 = ref (17*41) in
Printf.printf "%d\n" !r1;
Printf.printf "%d\n" !r2;
while !r1 > 0 do
  let r_max, r_min =
    if !r1 >= !r2 then r1, r2  else r2, r1
  in
  r_max := !r_max - !r_min
done;
Printf.printf "%d" !r2
```

Il y a deux façons de comparer les références :

- égalité structurelle `=` : compare le contenu des références,
- égalité physique `==` : compare la valeur des références.

Ainsi le code ci-dessous retourne le couple `(true, false)` car les deux références contiennent le même entier bien qu'elles ne désignent pas le même «emplacement mémoire».

```
let x = ref 1 in
let y = ref 1 in
x = y, x==y
```

1.2 Itérateur sur les fonctions

La fonction `List.fold_left` est très utile pour manipuler les listes sans avoir à créer de fonction auxiliaire pour itérer une construction. Elle suit le comportement suivant :

```
List.fold_left f a [x1;x2;...;xn] = f (... (f (f a x1) x2) xn
```

Elle vient avec son dual :

```
List.fold_right f a [x1;x2;...;xn] = f (... (f x2 (f x1 a)) ...) xn
```

Par exemple :

```
List.fold_left (+) 0 [1;2;3] = ((0 + 1) + 2) + 3
List.fold_right (+) 0 [1;2;3] = 1 + (2 + (3 + 0))
```

Question 1. – Programmer `rev` et `lenth` en utilisant `fold_left`.
– (Difficile) Programmer `fold_right` en utilisant `fold_left` (sans `let rec` !).

1.3 Liste d'associations

La fonction `List.assoc` et la fonction `List.assq` sont aussi très utile pour le TP. Si `l` est une liste de couple, `List.assoc x l` retourne le premier `y` tel que `(x',y)` $\in$ `l` et `x = x'`. Elle lève l'exception `Not_found` si elle ne trouve rien. Enfin, la fonction `List.assq` fait la même chose mais en utilisant l'égalité physique `==` à la place de `=`.

2 Un algorithme d'unification quasi-linéaire

Le but du TP est d'implémenter un algorithme d'unification. Un problème d'unification consiste en la donnée d'une liste d'égalité à satisfaire comme par exemple

$$\begin{aligned} x &= f(y,y) \\ f(y,z) &= f(f(t,t),t) \end{aligned}$$

Et une solution à un problème d'unification sera une façon d'instancier chacune des variables pour satisfaire toutes les équations du problème d'unification. Ici la solution la plus générale à ce problème d'unification est :

$$\begin{aligned} x &\mapsto f(f(t,t),f(t,t)) \\ y &\mapsto f(t,t) \\ z &\mapsto t \end{aligned}$$

2.1 Expression sans partage

Le fichier `expr.ml` contient les définitions

```
type var = string
type symbol = string

type t =
  | Var of var
  | Op of symbol * t list
```

qui permettent de représenter les expressions issues de la grammaire suivante :

$$e_1, ..., e_n := \quad x \quad | \quad f(e_1, ..., e_n)$$

Ainsi l'expression $g(x, f(x, y))$ est représentée par la valeur :

```
Op ("g", [Var "x"; Op ("f", [Var "x";Var "y"])])
```

Les fichiers `expr_lexer.mll`, `expr_parser.mly`, `expr_conv.ml` contiennent le nécessaire pour implémenter les fonction `string_of_expr` et `expr_of_string` qui permettent de convertir les expressions en chaîne de caractères et vice versa. Ainsi `(expr_of_string "g(x,f(x,y))")` retourne la valeur ci-dessus.

2.2 Expression avec partage

Si on utilise une représentation naïve des expressions, il arrive que les solutions aux problèmes d'unification occupent une taille exponentielle en la taille des équations du problème.

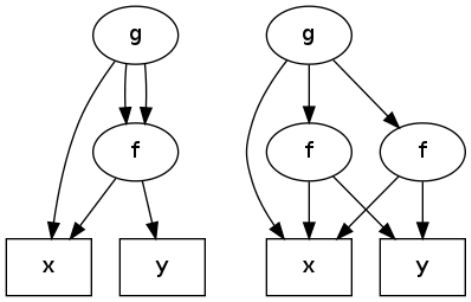
Question 2. Résoudre le problème d'unification suivant :

$$f(x_0, ..., x_{n-1}) = f(g(x_1, x_1), ..., g(x_n, x_n))$$

Quelle est la taille des solutions en fonction de n ?

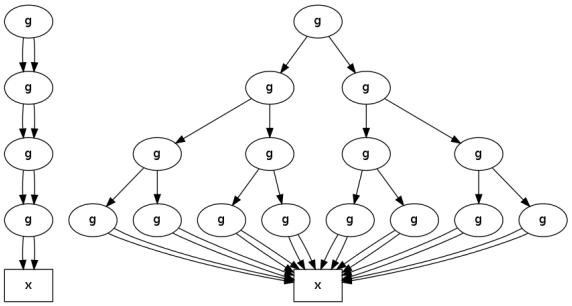
Ce genre de comportement peut être évité en implémentant intelligement la structure pour représenter les termes de façon à ne pas oublier que deux sous-termes sont issus d'une même duplication.

Ainsi, on representera les termes non plus comme des arbes mais comme des graphes qui peuvent contenir des cycles non-orientés. Par exemple, les deux graphes



représenteront le même terme $g(x, f(x, y), f(x, y))$.

Vous pouvez voir sur ces deux représentations de T_4 que le partage nous offre une représentation linéaire des T_i.



Ces dessins seront représentés en mémoire en utilisant les types définis dans le fichier `sharing.ml` :

```
type symbol = string
type var = string

type descr =
| Link of  t
| Repr of constr

and constr =
| Var
| Op of symbol  * t list

and t = descr ref
```

Un graphe sera représenté par un habitant du type `Sharing.t`, ce sera donc une référence. Par exemple, le code suivant donne une représentation de $g(f(x,y), f(x,y))$.

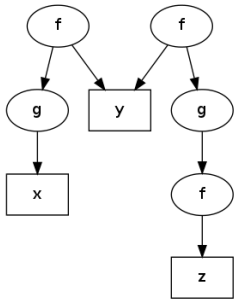
```
let x = ref Repr Var in
let y = ref Repr Var in
let fxy = ref Repr (Op (f, [x;y])) in
ref Repr (Op (g, [fxy;fxy]))
```

Notez que le nom des variables n'apparaît plus dans la structure de donnée. Ce nom pourra être utile dans la suite si on veut pouvoir afficher les termes. On utilisera donc des listes d'associations de type `(var * t) list` pour se souvenir du noms des variables. Ainsi dans l'exemple précédent, on utiliserait la liste `[("x",x);("y", y)]`.

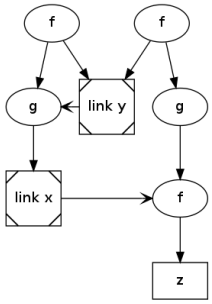
La première fonction à implémenter est un plongement de `Expr.t` dans `Sharing.t`. Elle prend en paramètre une liste de type `(var * t) list` qui contient la représentation des variables déjà défines et retourne à la fois l'objet converti (c'est la première composante) et une mise à jour de la liste des variables (elle est mise à jour dans le cas où on rencontre pour la première fois une variable durant la conversion).

Question 3. Implémentez la fonction `sharing_of_expr` dans le fichier `sharing.ml`

Le constructeur `Link` que nous n'avons pas encore rencontré va servir à créer des liens entre les termes que l'on veut unifier. Le but de la procédure `unify` va être de remplacer les variables par des liens pour satisfaire les contraintes d'unification. Par exemple, si on veut unifier les deux racines du graphe



on doit introduire les liens ci-dessous.



Les liens introduits lors de la procédure d'unification peuvent créer de longues chaînes qu'il va falloir parcourir afin de récupérer le `Repr` qui le termine.

Question 4. Implémenter la fonction `Repr` dans le fichier `sharing.ml`. Profitez-en pour lui faire «compresser les chemins» qu'elle parcourt (c-à-d réduire la chaîne en faisant pointer les liens directement vers le représentant récupéré).

Question 5. Implémentez la fonction `unify` dans le fichier `sharing.ml`.

3 Point d'entrée du programme

Le fichier `main.ml` est le point d'entrée de notre programme. Il commence par la génération de la liste d'exemples de problèmes à résoudre et se termine par la fonction `solve` qui lance la procédure d'unification sur la liste d'équations de chaque problème et affiche le résultat de l'unification.

```
let solve ((eqns, defined) as problem : Sharing.problem) =
  Printf.printf "Solving : \n%s\n" (string_of_problem problem);
  try
    List.iter (fun (x,y) -> unify x y) eqns;
    Printf.printf "Solution : \n";
    List.iter (fun (var, r) ->
      Printf.printf "%s -> %s\n" var
      (string_of_sharing_swapped defined r)) defined
  with Not_unifyable -> Printf.printf "Not unifyable.\n"
```

4 Test de cyclicité

Certains problèmes d'unification n'ont que des solutions infinies c'est le cas par exemple du problème $g(x, f(x)) = g(f(y), y)$ (la solution est de poser $x = y = f(f(f(...))))$. Dans de nombreuses applications de l'unification, ce genre de solutions sont rejetées.

Il n'est pas difficile de tester si une solution est cyclique, il suffit de faire un parcours en profondeur.

Question 6. Implémentez le test de cyclicité dans le fichier `sharing.ml`.