TP 3 : Unification optimisée

1 Rappels de `OCaml`

Cette section est facultative. Si vous êtes à l'aise sur les notions abordées, vous pouvez passer directement à la suite.

1.1 Référence et égalité physique

En `OCaml`, 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 (autrement dit, l'"emplacement mémoire" qu'elles désignent).

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 Itérateurs sur les fonctions

La fonction `List.map` est très utile pour appliquer une fonction à tous les membres d'une liste sans avoir à créer de fonction auxiliaire. Elle suit le comportement suivant :

```
List.map f [x1;x2;...;xn] = [f x1; f x2; ... ; f xn]
```

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 x1 ( f x2 (... (f xn a) ...))
```

Par exemple :

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

1. Programmez `rev` et `length` en utilisant `fold_left`.
2. (Difficile) Programmez `fold_right` en utilisant `fold_left` (sans `let rec` !).

1.3 Listes d'associations

La fonction `List.assoc` et la fonction `List.mem_assoc` pourront aussi être utiles pour le TP. Si `l` est une liste de couples, `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. Pour éviter de gérer les exceptions, on pourra utiliser `List.assoc_opt` (à partir d'`OCaml` 4.05) qui renvoie `Some y` si elle trouve quelque chose et `None` sinon ou `List.mem_assoc` qui renvoie `true` si un tel `y` existe et `false` sinon.

Il existe des variantes de `List.assoc`, `List.assoc_opt` et `List.mem_assoc` qui sont `List.assq`, `List.assq_opt` et `List.mem.assq` qui font la même chose mais en utilisant l'égalité physique == à la place de =.

2 Vers un algorithme plus efficace

Dans une version précédente du TP, on a implémenté un premier algorithme d'unification en suivant à la lettre celui vu en cours, à savoir celui de la figure 1.

1. Résolvez le problème d'unification suivant en utilisant le premier algorithme que vous avez implémenté :

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

2. Quelle est la taille des solutions en fonction de n ?

Pour améliorer l'efficacité de l'implémentation de l'algorithme d'unification, on fait deux modifications :

Algorithme d'unification

relation $\longrightarrow$ entre états, où un état est soit $(\mathcal{P}, \sigma)$, soit $\bot$

- $\bot \not\longrightarrow$ et $(\emptyset, \sigma) \not\longrightarrow$
- on considère $(\{t \stackrel{?}{=} t'\} \uplus \mathcal{P}, \sigma)$
  - **(1)** $(\{f(t_1, \ldots, t_k) \stackrel{?}{=} f(u_1, \ldots, u_k)\} \uplus \mathcal{P}, \sigma)$
    $\longrightarrow (\{t_1 \stackrel{?}{=} u_1, \ldots, t_k \stackrel{?}{=} u_k\} \cup \mathcal{P}, \sigma)$
  - **(2)** $(\{f(t_1, \ldots, t_k) \stackrel{?}{=} g(u_1, \ldots, u_n)\} \uplus \mathcal{P}, \sigma) \longrightarrow \bot$ (avec $f \neq g$)
  - **(3)** $(\{X \stackrel{?}{=} t\} \uplus \mathcal{P}, \sigma) \longrightarrow (\mathcal{P}[t/X], [t/X] \circ \sigma)$ si $X \notin \mathrm{vars}(t)$
    où $\mathcal{P}[t/X] = \{u_1[t/X] \stackrel{?}{=} u_2[t/X],\ u_1 \stackrel{?}{=} u_2 \in \mathcal{P}\}$
  - **(4)** $(\{X \stackrel{?}{=} t\} \uplus \mathcal{P}, \sigma) \longrightarrow \bot$ si $X \in \mathrm{vars}(t)$ et $t \neq X$
  - **(5)** $(\{X \stackrel{?}{=} X\} \uplus \mathcal{P}, \sigma) \longrightarrow (\mathcal{P}, \sigma)$

FIGURE 1 – L'algorithme d'unification vu en cours

— on travaille avec des termes où l'on partage les variables. Les variables sont implémentées à l'aide de pointeurs. Les termes ont alors une structure de DAG. Cela permet de ne pas transporter une substitution courante ; au contraire, l'algorithme d'unification procède par mise à jour des pointeurs. On notera, informellement, $x \mapsto t$ quand la variable x pointe vers le terme t.

— on ne fait pas le test "`X∈vars(t)`" dans le cas (4). On fonce tête baissée, et on vérifie après coup si la solution est correcte via un test de cyclicité.

3 Termes avec partage de variables

Voici la représentation que l'on vous propose pour les termes et problèmes d'unification avec variables partagées :

```
type symbol = string

type t =
  | Var of (t option) ref
  | Op of symbol  * t list

type var = string
type problem = ((t * t) list) * ((var * t) list)
```

Un problème d'unification est maintenant un ensemble d'équations à satisfaire, certes, mais au sein desquelless les variables ont toutes été remplacées par une référence ; la deuxième partie, `(var * t) list` garde en mémoire l'ensemble des variables déjà rencontrées et définies.

Pour représenter le terme $f(x_1, x_2, x_3)$, on procèdera ainsi :

```
let vx1 = ref None
let vx2 = ref None
let vx3 = ref None
let t1 = Op("f", [Var vx1; Var vx2; Var vx3])
```

et pour représenter $f(y_1, y_2, y_1)$, on fait

```
let vy1 = ref None
let vy2 = ref None
let t2 = Op("f", [Var vy1; Var vy2; Var vy1])
```

Notez qu'il est important de faire un `ref None` à chaque fois, pour obtenir des valeurs différentes au sens de ==.

Lorsque l'on exécute l'algorithme d'unification, plutôt que d'étendre la substitution avec un couple (x, t), on fait pointer le `ref` de la variable x vers `Some t`.

Les fichiers `Makefile`, `expr.ml`, `expr_conv.ml`, `expr_lexer.mll` et `expr_parser.mly` sont fournis et sont les mêmes que dans la version précédente. On donne aussi `sharing_conv.ml` qui implémente des procédures pour transformer un terme avec partage de variables en chaîne de caractères, `sharing_expr.ml` qui contient les définitions des types avec partage de variables, et `main.ml` où l'on trouvera des exemples sur lesquels faire des tests, ainsi que les procédure permettant l'application de l'algorithme d'unification aux exemples. Le seul fichier que vous avez à modifier est `sharing_unif.ml`, éventuellement en utilisant ce qui est définie dans `sharing_expr.ml`.

1. Écrivez une fonction de traduction d'un terme sans partage vers `t` :

```
sharing_of_expr: (string * t) list -> Expr.t -> t * (string * t) list}
```

Cette fonction prend une liste de couples associant à chaque identificateur de variable son terme, et la met à jour lorsqu'elle rencontre de nouveaux noms de variable. Cette liste pourra être utile pour traquer les bugs dans votre code, et pour afficher la substitution calculée par l'algorithme d'unification (cf. la procédure `solve` dans `main.ml`).

4 Unifier en manipulant des pointeurs

Lorsque l'on unifie, chaque fois que l'on enrichit la substitution courante avec une association $x \mapsto t$, on met à jour le pointeur que contient la représentation de x. En cours d'exécution, la structure des pointeurs décrit la substitution courante.

Ne pas s'embarquer dans les cycles.

1. Réfléchissez au problème d'unification

$$x = f(y) \quad y = f(x) \quad z = f(t) \quad t = f(z) \quad x = z.$$

On installe un certain nombre de pointeurs, en particulier, après quelques étapes :
— x et z pointent vers $f(y)$ et $f(t)$ respectivement,
— y et t pointent vers $f(x)$ et $f(z)$ respectivement.
Au moment d'unifier x et z, il faut prendre garde à ne pas se faire happer naïvement par la récursion, qui conduit à unifier y et t, puis x et z, et à boucler indéfiniment.

Ne pas créer de cycles entre variables. Une substitution identifie une variable soit à une autre variable, soit à un "terme construit", soit à rien.

On s'attachera à ne pas créer de cycles entre variables : pas de $x \mapsto y, y \mapsto x$ par exemple. Ainsi, les pointeurs entre variables auront une structure de forêt, les racines des arbres étant des variables pointant vers rien ou bien vers un terme construit.

2. Définissez une fonction `repr : t -> t` qui associe à un terme son représentant au sens suivant :
   - l'identité pour un `Op (f,l)` et pour un `Var p` avec `!p = None`,
   - le représentant du terme vers lequel pointe `p` pour `Var p` avec `!p <> None`.

   Notez que `repr` ne peut pas rendre du `Var (Some _)`.

   Réfléchissez au lien entre la terminaison de cette fonction et la structure en forêt mentionné ci-dessus.

En revanche, puisque l'algorithme ne fait pas de test d'apparition, on pourra engendrer des substitutions contenant des cycles qui passent à travers des constructeurs, comme $x \mapsto f(x)$, ou $x \mapsto f(y), y \mapsto x$.

Faites bien attention à la notion d'égalité (physique ou non) que vous utilisez dans les deux dernières questions.

3. Implémentez l'algorithme d'unification de deux termes décrit informellement dans la section 2. Veuillez prendre garde à ne pas créer de cycles entre variables et à éviter le cas de la section précédente.
4. Implémentez un test de cyclicité en sortie de l'algorithme d'unification qui aura pour effet de rejeter les solutions contenant des cycles qui passent à travers des constructeurs.