TP 3 : Unification simple

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

$$\begin{array}{rcl} x & \stackrel{?}{=} & f(y,y) \\ f(y,z) & \stackrel{?}{=} & f(f(t,t),t). \end{array}$$

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 est :

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

1 Expressions

Le fichier `expr.ml` contient les définitions des types pour notre problème :

```
type var = string
type symbol = string

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

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

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

$$e_1, \ldots, e_n \ ::= \ x \mid f(e_1, \ldots, e_n)$$

ainsi que les problèmes d'unification associés.

L'expression $g(x, f(x,y))$ est représentée par la valeur suivante dans notre code :

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

Alors qu'en cours chaque constante a une arité (cas `Op`), ici on ne fait pas cette hypothèse. Cela engendrera des cas d'échec dans l'algorithme d'unification si on essaie d'unifier deux termes ayant la même constante à la racine mais des nombres de sous-arbres différents.

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

Description des fichiers OCaml fournis. Les fichiers `expr_lexer.mll`, `expr_parser.mly` et `expr_conv.ml` contiennent le nécessaire pour implémenter les fonctions `string_of_expr` et `expr_of_string` qui permettent de convertir les expressions en chaînes de caractères et vice versa. Ainsi, l'exécution de `(expr_of_string "g(x,f(x,y))")` retourne la valeur ci-dessus.

Dans ce TP, on ne vous demande pas de modifier, comprendre ni même lire ces différents fichiers. Ils vous permettront seulement de lancer des tests sur votre fonction d'unification. **Globalement, les seuls fichiers qui vont vous intéresser sont `expr.ml`, `unif.ml` et `main.ml`, et le seul fichier qu'on vous demande de modifier est `unif.ml`.**

Le `main.ml` lance automatiquement votre procédure d'unification sur une liste d'exemples définie au début du fichier. Pour tester votre programme, il vous suffit donc d'exécuter `make run` (ou simplement `make` pour seulement recompiler) (vous pouvez faire appel à la personne qui encadre le TP si vous avez besoin d'aide sur ce point).

2 Implémenter l'algorithme d'unification

Le but de ce TP est de compléter le fichier `unif.ml` afin d'obtenir une procédure d'unification, et de la vérifier sur les exemples présents dans le fichier `main.ml`. On travaillera avec les types introduits dans le fichier `expr.ml`, ainsi qu'avec le type `subst` défini au début du fichier `unif.ml`.

1. Implémentez la fonction `appear` qui, à une une variable de type `var` et un terme de type `t`, renvoie un booléen indiquant si la variable apparaît dans le terme ou non.
2. Implémentez la fonction `replace` telle qu'étant donnés une variable `x` de type `var` et

deux termes new_x et term de type t, replace (x, new_x) term renvoie le terme term dans lequel toutes les occurences de x ont été remplacées par new_x.

Pour écrire replace, vous pouvez utiliser la fonction `List.map` dans le cas `Op`.

3. Implémentez la procédure d'unification unify : problem → subst permettant de résoudre un problème d'unification (de type problem) en renvoyant une substitution solution de type subst (dans le cas ou le problème n'a pas de solution, on lèvera une exception). Vous pouvez vous référer à l'algorithme décrit dans la Figure 1 pour définir votre procédure. Faites particulièrement attention au cinquième cas de l'algorithme et à la composition des substitutions.
4. Vérifiez votre algorithme sur les exemples donnés dans le fichier `main.ml`, voire sur d'autres exemples que vous pouvez rajouter au début de ce fichier.

3 Complexité de l'algorithme

1. Dans le fichier `main.ml`, l'exemple big n correspond au problème d'unification suivant :
$$f(x_0, ..., x_{n-1}) = f(g(x_1, x_1), ..., g(x_n, x_n))$$
Quelle est la solution donnée par votre algorithme sur ce problème ?
2. En déduire une borne inférieure de la complexité de votre algorithme.

On peut grandement réduire cette complexité en adoptant une représentation plus efficace pour les termes. La deuxième variante de ce TP développe plus en détail cette approche.