Programmation 1 – TP n° 10
Unification

1 Rappels

Le but de ce TP est d'implémenter l'agorithme d'unification du premier ordre de *Martelli-Montanari* vu en cours.

On travaille sur des *termes* définis inductivement à partir de variables et d'un ensemble de symboles de fonctions. Par exemple, si l'ensemble des symboles de fonctions est $\{0_0, S_1, f_2\}$ (les arités sont portées en indices), et que les variables sont dénotées par $x, y, z \ldots$ on aura les termes suivants :

- termes clos : $0,\ S(0),\ S(S(0)),\ f(0, S(0)),\ f(f(0,0), f(f(0,0)))\ldots$
- termes ouverts : $x,\ S(x),\ S(S(y)),\ f(x, S(0)),\ f(f(x,x), f(f(y,y)))\ldots$

Un *unificateur* de deux termes u et v, c'est une substitution σ des variables vers les termes, telle que $\sigma(u) = \sigma(v)$. Un unificateur *principal* de deux termes u et v est un unificateur σ tel que tout autre unificateur ρ de u et v soit de la forme $\rho = \rho'\sigma$.

Dans notre cas (*premier ordre*), les unificateurs principaux sont uniques – modulo les renommages de variables. Pour les trouver, on génère à partir de deux termes u et v, un ensemble de contraintes d'égalité de la forme $\{x_i = w_i\}$, où les x_i sont des variables et les w_i des termes. Par exemple, pour $u = f(x, f(x,y)),\ v = f(S(y), f(y,x))$, on aura l'ensemble de contraintes $\{x = S(y), x = y, y = x\}$.

On va ensuite *travailler* cet ensemble de contraintes jusqu'à ce qu'une erreur se produise (les termes ne sont pas unifiables) ou que l'on obtienne une substitution, correspondant à l'unificateur principal. Un tel ensemble vérifie les propriétés suivantes.

- Chaque variable n'est définie qu'une seule fois. Dans l'exemple y l'est, mais pas x.
- La définition d'une variable (le terme qu'on trouve à sa droite) n'est pas *récursive*. Dans l'exemple, x est récursif car, si on déroule la définition de y, on obtient $x = S(x)$.

Le deuxième point se nomme *Occurs-Check* et caractérise les solutions *finies*. L'algorithme terminera même sans cette règle.

Les règles permettant de travailler l'ensemble de contraintes seront rappelées par la suite.

2 Ensemble de Tarjan

Commencez par récupérer sur le site une implémentation des ensembles de Tarjan ayant la signature suivante :

```
(* ensembles d'elements de 'a, associes a des valeurs de type 'b *)
    type ('a,'b) t
(* creation *)
    val create: unit -> ('a,'b) t
```

```
(* ajout d'un singleton *)
    val add: ('a,'b) t -> 'a -> 'b -> unit
(* remplacement de la valeur associee a un ensemble existant *)
    val replace: ('a,'b) t -> 'a -> 'b -> unit
(* recherche de la valeur associee a un ensemble existant *)
    val find: ('a,'b) t -> 'a -> 'b option
(* fusion de deux ensembles existant, la valeur associee a l'union
est le 'b donne *)
    val merge: ('a,'b) t -> 'a -> 'a -> 'b -> unit
(* test d'equivalence de deux variables *)
    val equiv: ('a,'b) t -> 'a -> 'a -> bool
(* iterations des classes d'equivalence (par leur representants) *)
    val iter_repr: ('a -> 'b -> unit) -> ('a,'b) t -> unit
(* conversion de la table en une liste d'association *)
    val to_list: ('a,'b) t -> ('a * 'b) list
```

Cette structure nous permettra de manipuler des classes d'équivalence de variables annotées par des termes.

3 Unification

3.1 Représentation des termes

Afin de rester abstrait, notre représentation des termes est paramétrée par un ensemble de symboles de fonction :

```
type var = char

type 'a term =
  | Var of var                          (* variable *)
  | Constr of 'a * 'a term list         (* constructeur *)
```

On néglige ici les questions d'arité qui ne sont pas importantes pour l'unification : deux listes doivent avoir la même longueur pour être unifiées.

3.2 Génération des contraintes

Lors de l'unification de deux termes $s \equiv t$, la première étape consiste à générer des contraintes d'égalité entre variables et termes à partir des trois règles suivantes :

$$\frac{P \uplus \{f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)\}}{P \uplus \biguplus_{1 \leqslant i \leqslant n} \{u_i \equiv v_i\}} Decompose \qquad \frac{P \uplus \{f(\ldots) \equiv g(\ldots)\} \qquad f \neq g}{\text{fail}} Conflict$$

$$\frac{P \uplus \{u \equiv x\} \qquad u \text{ n'est pas une variable}}{P \uplus \{x \equiv u\}} Switch \qquad \frac{P \uplus \{u \equiv u\}}{P} Simplify$$

Comme la phase de résolution demande de rajouter des contraintes, il est plus malin de définir une fonction `add_constraints` de type
`(var * 'a term) list -> 'a term -> 'a term -> (var * 'a term) list.`

Q 3.1 Écrivez cette fonction de telle sorte qu'elle lève, en cas de problème, les exceptions `Fail_Constr` et `Fail_Arite`.

3.3 Résolution

Il faut désormais réécrire l'ensemble obtenu par `add_constraints` de manière à obtenir la substitution finale. Ici, nous allons éviter de *dérouler* cette substitution pour des questions de complexité (essayez donc avec la substitution $\{x = f(y, y),\ y = f(z, z),\ z = f(w, w),\ t = x,\ u = t\}$).

Afin de stocker de manière *condensée* cette substitution, nous allons la considérer comme un ensemble de Tarjan.

```
resolve: (var * 'a term) list -> (var, 'a term) Tarjan.t
```

La fonction `resolve` prend une liste d'égalités entre variables et termes, et renvoie, si possible, une partition des variables telle qu'à chaque élément de la partition corresponde un terme. Dans les cas où l'unification est impossible, elle lèvera des exceptions de la forme `Failure "message adéquat"`. Elle ne s'occupe pas de la règle *Occurs-Check* qui détecte les occurrences récursives de variables. Voici une ébauche des règles à appliquer :

- Créer un ensemble de Tarjan vide, appelé *table*, que l'on modifiera au fur et à mesure.
- Si la liste de contraintes est vide, renvoyer la table.
- Sinon, soit $x = u$ l'égalité en tête de liste,
  - si u est une variable ($u = y$) :
    - si x (resp. y) n'existe pas dans la table, la rajouter à la classe de y (resp. x) (si aucun des deux n'existe, les rajouter tous les deux, dans une même classe)
    - si les deux existent :
      - si elles sont équivalentes, passer à la suite,
      - si l'une d'elles est associée à une variable, unifier leurs classes,
      - si les deux sont associées à de *vrais* termes (avec au moins un constructeur) notés u_x et u_y, unifier les deux classes, en prenant u_x comme valeur associée ; et rajouter les contraintes engendrées par l'égalité $u_x = u_y$.
  - sinon (u est un vrai terme) : si x n'existe pas, le rajouter en l'associant à u, sinon :
    - s'il est associé à une variable, remplacer cette association par u,
    - s'il est associé à un terme u', rajouter les contraintes engendrées par l'égalité $u = u'$.
- Continuer avec la liste de contraintes restantes (qui a éventuellement été augmentée).

Q 3.2 Écrivez cette fonction.

Q 3.3 Testez votre fonction avec un langage d'expressions arithmétiques (`O | S | +`).

3.4 Occurs-Check

Q 3.4 Que renvoie votre fonction sur l'exemple $x \equiv Sx$?

Il manque une règle qui vérifie qu'une variable n'apparaît pas dans le terme qui la définit, et que la solution est donc finie.

Q 3.5 Implantez cette vérification. Elle sera appelée une fois une solution trouvée.

```
occurs_check: (var, 'a term) Tarjan.t -> unit
```