Fiche de TD-TP no. 4

Exercice 1. Voici trois façons différentes de définir le type `Image` :

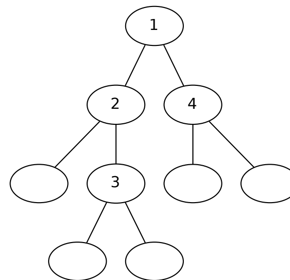
```
type Image = [[Int]]
data Image = Image [[Int]]
newtype Image = Image [[Int]]
```

Rappelez la différence entre les mots clés `type`, `data`, et `newtype`. A votre avis, quell'est la meilleure façon de définir le type `Image` ? Expliquez pourquoi.

Exercice 2. Considérez la variante suivante du type récursif des arbres binaires, défini par :

```
data ArbreBin = Feuille | Noeud ArbreBin Int ArbreBin
```

1. Écrivez une expression Haskell, de type `ArbreBin`, qui représente l'arbre



2. Dessinez l'arbre binaire représenté par `a1` dans la définition Haskell

```
a1 = Noeud (Noeud Feuille 4 Feuile)
     3
     (Noeud (Noeud Feuille 5 Feuile) 6 Feuile)
```

3. Proposez trois autres expressions Haskell de type `ArbreBin` et dessinez les arbres qu'elles dénotent.
4. Définissez les fonctions suivantes :

`taille : ArbreBin -> Int`	compte le nombre des noeuds dans un arbre binaire,
`maxA : ArbreBin -> Int`	l'étiquette maximum d'un arbre binaire,
`minA : ArbreBin -> Int`	l'étiquette minimum d'un arbre binaire.

   Comment modifier les fonctions `maxA` et `minA` si on veut que `maxA Feuille` $= -\infty$ et `minA Feuille` $= +\infty$?
5. Un arbre binaire est un *arbre de recherche* si l'étiquette de chaque noeud est plus grande des toutes les étiquettes dans le sous-arbre à gauche, et plus petite des toutes les étiquettes dans le sous-arbre à droite. Définissez un prédicat Haskell `aRecherche :: ArbreBin -> Bool` qui répond vrai à un arbre donné ssi il s'agit d'un arbre de recherche.
6. Un arbre binaire est équilibré si, pour chaque noeud, la taille du sous-arbre à gauche et celle du sous-arbre à droite différent au plus de 1. Définissez un prédicat Haskell `aEquilibre :: ArbreBin -> Bool` qui répond vrai à un arbre donné ssi il s'agit d'un arbre équilibré.
7. Est ce que les fonctions `aRecherche` et `aEquilibre` sont performantes ? Comment modifier la définition du type `arbreBin` pour améliorer la performance de ces fonctions ?
8. La plus part des algorithmes sur les arbres binaires peuvent se spécifier comme suit :

- dans le cas d'une feuille, on retourne un valeur donné `v` ;
- pour le cas d'un noeud, on calcule la valeur de cet arbre à laide d'une fonction `f` de trois arguments : un entier est les deux valeurs calculés de façon récursive avec le sous-arbre à gauche et le sous-arbre à droite.

  Montrez comment
  - les fonctions `taille`, `maxA` et `minA`,
  - le calcul de la somme de toutes les étiquettes d'un arbre,
  - le calcul de l'hauteur d'un arbre (longueur de la plus longue branche)

  peuvent se faire/définir selon ce schéma de calcul : qui sont `v` et `f` ?

9. Définissez une fonction

   `foldArbre :: a -> (a -> Int -> a -> a) -> ArbreBin -> a`

   qui capture le schéma de récursion qu'on vient de voir.

Exercice 3. Voici une possible déclaration de la classe `Set` :

```
class Set s where
    appartient :: Eq a => a -> s a -> Bool
    ajouter :: Eq a => a -> s a -> s a
    enlever :: Eq a => a -> s a -> s a
    pick_one :: Eq a => s a -> a
    vide :: s a
    est_vide :: s a -> Bool
    union :: Eq a => s a -> s a -> s a
    intersection :: Eq a => s a -> s a -> s a
    difference :: Eq a => s a -> s a -> s a
```

Proposez deux types – un basé sur les listes, l'autre sur les arbres binaires – et instanciez cette classe avec ces types.

En TP

Dans cette séance de TP nous fairons les premiers pas pour accomplir le projet (voir page web du cours), qui vous demande d'implémenter l'algorithme d'unification.

Une signature est un couple $\mathcal{S} = (\Omega, ar)$ où Ω est un ensemble et $ar : \Omega \to \mathbb{N}$. On dit que Ω est l'ensemble des symboles de functions et que, pour $f \in \Omega$, $ar(f)$ est l' arité de f.

Soit X un ensemble de variables. L'ensemble $\mathcal{T}_{\mathcal{S}}(X)$—des termes sur la signature $\mathcal{S} = (\Omega, ar)$ avec variables dans X—est défini par induction comme suit :

- pour $x \in X$, $x \in \mathcal{T}_{\mathcal{S}}(X)$;
- si $f \in \Omega$, $ar(f) = n$, et $t_1, \ldots, t_n \in \mathcal{T}_{\mathcal{S}}(X)$, alors $f(t_1, \ldots, t_n) \in \mathcal{T}_{\mathcal{S}}(X)$;
- rien d'autre est un terme.

Une substitution est une fonction $\sigma : X \to \mathcal{T}_{\mathcal{S}}(X)$ telle que l'ensemble $E(\sigma) = \{ x \in X \mid \sigma(x) \neq x \}$ est fini.

Exercice 4. Définissez dans un script Haskell

- le type des variables,
- le type des symboles de fonctions,
- le type d'une signature,
- le type des termes,
- le type des substitutions.

Tous ces types seront des instances de la classe `Show` (et de la classe `Eq`, si possible).

La difficulté majeure de cet exercice est la définition du type des termes. Pour cela, on peut lire la définition mathématique de l'ensemble des termes comme spécifiant qu'il y a seulement deux façon pour *construire* un terme : à partir d'une variable, ou bien à partir d'un symbole de fonction et d'une liste de termes (on fait abstraction de la condition $ar(f) = n$).

Aussi, pour bien définir le type des substitutions, considérez que l'ensemble $E(\sigma)$ est fini, de façon qu'une substitution puisse être représentée par un objet fini.

Exercice 5. Ajoutez, dans votre script Haskell, la définition d'une fonction `variables` qui calcule les variables qui apparaissent dans un terme.

Nous pouvons définir ainsi l'application d'une substitution σ à un terme t, notée $t \cdot \sigma$ ou $t\sigma$, comme suit :

- $x \cdot \sigma := \sigma(x)$,
- $f(t_1, \ldots, t_n) \cdot \sigma := f(t_1 \cdot \sigma, \ldots, t_n \cdot \sigma)$.

Exercice 6. Ajoutez, dans votre script Haskell, la définition de cet opérateur d'application.

Enfin, si σ et τ sont des substitutions, nous pouvons définir leur composition, notée $\tau \circ \sigma$, comme suit :

$$(\tau \circ \sigma)(x) := \sigma(x) \cdot \tau\,,$$

de façon qu'on a la loi suivante :

$$x \cdot (\tau \circ \sigma) := (x \cdot \sigma) \cdot \tau\,.$$

Exercice 7. Ajoutez, dans votre script Haskell, la définition de l'opérateur de composition.

Exercice 8. Complétez votre script pour définir un module complet `Term.hs` selon les spécifications du « listing » suivant :

Term

```
module Term (
    Var, FuncSymb, Signature, Term, varToTerm, constructTerm, variables,
    sig, Substitution, identity, (*!), (@@)
  ) where
```

`data Var`

Votre représentation concrète et préférée des variables

```
instance Eq Var
instance Show Var
```

`data FuncSymb`

Votre représentation concrète et préférée des symboles de fonctions

```
instance Eq FuncSymb
instance Show FuncSymb
```

`type Signature = [(FuncSymb, Int)]`

Nous représentons une signature comme une liste de paires, symbole de fonction plus son arité

`data Term`

Le type des termes (dont les variables sont de type v)

`instance Show Term`

`varToTerm :: Var -> Term`

Transtypage une variable vers un terme

`constructTerm :: String -> [Term] -> Term`

Construire un terme à partir d'un symbole de fonction et une liste de termes

`variables :: Term -> [Var]`

Les variables dans un terme donné

`sig :: Term -> Signature`

On récupère la signature d'un terme

`data Substitution`

Le type des substitutions (avec variables de type v)

`instance Show Substitution`

`identity :: Substitution`

La substitution qui envoie x vers x

`(*!) :: Term -> Substitution -> Term`

Application d'une substitution à un terme

`(@@) :: Substitution -> Substitution -> Substitution`

Composition de substitutions

Exercice 9. L'implantation du type des variables et du type des symboles de fonction étant assez arbitraire, pouvez vous redéfinir le type de termes en le paramétrant par deux variables de type, `v` (pour les variables) et `f` (pour les symboles de fonctions) :

```
data Term v f = ...
```