Les types sommes récursifs

Définition

Un type somme est récursif si son identificateur apparaît dans sa définition. Cela ne nécessite pas de syntaxe spéciale, mais une attention particulière.

Les types récursifs permettent de définir facilement des objets "évolutifs“ ou des collections. Par exemple

• Une peinture c’est du jaune, du rouge du bleu ou un mélange de peinture :

```
type peinture = Bleu|Jaune|Rouge
    | Melange of peinture*peinture;;
```

• Une bibliothèque c'est vide ou ce qu'on obtient en ajoutant un livre à une bibliothèque.

```
#type livre = Conte|Roman|Nouvelle;;
#type biblio = Vide | Ajout of livre*biblio;;
```

Bien sûr le meilleur traitement pour un type récursif est souvent lui-même récursif :

```
let rec nb_livres = fun
Vide ->0
|(Ajout( _,bib))->1+nb_livres(bib);;
```

• un arbre binaire c'est une feuille ou un noeud constitué de deux sous arbres. en prenant des valeurs entières par exemple :

```
type arbre = Feuille of int| Noeud of int*arbre*arbre
```

La somme des valeurs des sommets d'un arbre binaire entier est alors obtenue par :

```
let rec somme = fun
(Feuille n)->n
|(Noeud(n,t1,t2))->n+somme(t1)+somme(t2);;
```

Les types produit

Définition

Les types produits (aussi appelés enregistrements ou types "ou") permettent de regrouper dans un même type plusieurs objets différents appelés **champs** *de l'enregistrement, désignés par des noms appelés* **étiquettes de champs**.

La syntaxe est la suivante :

```
type id={c_1 : t_1 ; ... ; c_n : t_n}
```

où les C_i sont des identificateurs (les étiquettes de champs) et les t_i des types.

utilisation d'un type produit

On utilise alors un objet de type *id* par la syntaxe :

```
{c_1 = exp_1; ... ; c_n = exp_n}
```

où les exp_i sont des expressions de type t_i.
Remarques : 1. L'ordre des étiquettes n'importe pas mais il faut impérativement remplir tous les champs.
2. On provoque souvent des erreurs en définissant un nouveau type utilisant une étiquette de champs déjà utilisée. L'ancien type utilisant cette étiquette n'est alors plus utilisable.
3. Cette fois-ci les étiquettes de champs doivent commencer par des minuscules !

exemples

```
# type complexe = { re : float ; im : float};;

#let z = {re=1. ; im=0.};;
val z : complexe = {re = 1.0; im = 0.0}

#type livre = {titre:string;année:int;auteur:string};;
#let chat={titre ="Le chateau";année=1935;auteur="Kafka"};;
val chat : livre =
   {titre = "Le chateau"; année = 1935; auteur = "Kafka"}
```

Accès aux champs d'un type produit, fonctions

```
#z.im;;
- : float = 0.0

#chat.auteur;;
- : string = "Kafka"

#let l_auteur= fun  l->l.auteur;;
val l_auteur : livre -> string = <fun>

#let meme_auteur = fun  (l,k)->l.auteur=k.auteur;;
val meme_auteur : livre * livre -> bool = <fun>

#let dix9eme = fun  l->l.année>=1800 & l.année<1900;;
```

Les fonctions sont couramment définies par filtrage, les constructeurs de type produits { et } étant autorisés dans les filtres :

```
#let conjugué= fun  {re=a;im=b}->{re=a;im= -.b};;
val conjugué : complexe -> complexe = <fun>

#let auteur = fun  {auteur=a;année=_;titre=_}->a;;
```

Exercice : mélanges de types somme et produit

On définit les types :

```
#type monome={coeff:float;deg:int};;

#type polynome=Plein of int* float list
               |Creux of monome list;;
Le type polynome est défini.

#let p = Plein(4,[1.;0.;3.;0.;5.]);;
val p : polynome = Plein (4, [1.0; 0.0; 3.0; 0.0; 5.0])
```

Ecrire une fonction de conversion entre les représentations pleines et creuse d'un polynôme.

```
#let ajout = fun
(a,Creux l)->Creux (a::l);;
val ajout : monome * polynome -> polynome = <fun>

#let rec pleinVersCreux= fun
(Plein (0,[x]))-> if x = 0. then Creux []
                  else Creux [{coeff=x;deg=0}]
|(Plein (n,a::l))-> if a = 0. then
                       pleinVersCreux (Plein (n-1,l))
else ajout({coeff=a;deg=n},pleinVersCreux(Plein (n-1,l)));


          Characters 27-273:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Creux _
val pleinVersCreux : polynome -> polynome = <fun>
```

types polymorphes

Il est parfois utile de définir soi-même des types polymorphes. De tels types peuvent être déclarés à l'aide des variables polymorphes

```
'a, 'b
```

Par goût d'enfoncer les portes ouvertes, nous pouvons par exemple redéfinir le type liste :

```
#type 'a liste =
  Liste_vide
| Cons of 'a*'a liste;;
```

...et utiliser alors des listes d'entiers ou de chaînes de caractères.

```
#Cons (1,Liste_vide);;
- : int liste = Cons (1, Liste_vide)
#Cons("a",Cons("b",Liste_vide));;
- : string liste = Cons ("a", Cons ("b", Liste_vide))
```

Complément sur les types : Les contraintes de type

Définition

Lorsqu'on souhaite éviter le polymorphisme d'une fonction, on peut contraindre le type d'une expression à l'aide de la syntaxe

```
(expr : type)
```

.

Exemple : la comparaison de deux éléments :

```
# let egal = fun (x,y)->x=y;;
val egal : 'a * 'a -> bool = <fun>
```

peut être typée par

```
let egal = fun  ((x : int),(y : int)) -> x=y;;
val egal : int * int -> bool = <fun>
```

Différence Caml Light - O'Caml

Définition

Certaine fonctions appartenant au noyau Caml Light sont dans des bibliothèques en O'Caml. Il faut donc ouvrir ces bibliothèques pour les utiliser.

Exemple : La bibliothèque String :

```
# length "bonjour";;
Characters 0-6:
Unbound value length
# open String;;
length "bonjour";;
# - : int = 7
```

Bibliothèques

La bibliothèque List :

```
# hd[1;2;3];;
Characters 0-2:
Unbound value hd

# open List;;
#hd[1;2;3];;
- : int = 1
# tl[1;2;3];;
- : int list = [2; 3]
```

Bibliothèques

Si on ne souhaite pas ouvrir toute la bibliothèque mais seulement utiliser une fonction, on peut utiliser la syntaxe pointée.

```
#length[1;2;3];;
- : int = 3
# length "bonjour";; (* La fonction length de List a écras
Characters 7-16:
This expression has type string but is here used with type
# String.length "bonjour";;
- : int = 7
# List.length[1;2;3];;
- : int = 3
```