Types Construits

6 novembre 2023

Plan

Administratif

Rendu du DS

Construire son propre type en OCaml

Administratif

J'ai demandé à changer la salle des colles du mardi matin pour la B309. Il reste à confirmer si le changement a pu être fait.

Rendu du DS

Rendu du DS

Moyenne	Q1	Médiane	Q3	Max
13,0	10,9	12,8	15,6	19,6

Le niveau est assez satisfaisant, mais il y a beaucoup d'approximations sur les démonstrations et sur le code qui seront punis dans les DS suivants.

Quelques remarques sur la composition

- Vous ne pouvez pas faire un brouillon pour tous vos codes, mais il est utile d'avoir du brouillon parfois ;
- Que ce soit en cours, en TP, ou en DS d'informatique, vous pouvez aller aux toilettes sans en demander la permission (vous serez accompagnés et vous devrez demander la permission au concours cependant) ;
- Pensez à ramener votre propre matériel, les copies doubles et les brouillons seront fournis au concours, mais vous ne pourrez pas faire passer du matériel de bureau entre les tables durant l'épreuves.

Questions en DS

- Des indications figurent sur le sujet, ici en particulier, il y avait le protocole en cas d'erreur, ainsi que des informations sur l'utilisation de codes ou réponses précédentes ;
- En pratique, je n'ai rien laissé passé comme question lors de ce sujet, il y aura moins d'erreurs ou d'imprécisions au concours, mais vous aurez encore moins d'aide ;
- Je ne peux absolument pas vous aider sur les questions de cours, ou de programmation, vous avez accès aux outils dans le préambule dont vous pourriez avoir besoin ;
- S'il y a un gros contre-sens, ou une grosse source d'erreur, je vous en informerai à voix haute à toute la classe ;
- Vous n'êtes pas limités à ces outils, mais il est préférable d'utiliser vos propres fonctions dans la mesure du possible.

Remarques sur le contenu

- Vous aurez parfois des choses idiotes à faire dans les sujets qui ne rapportent pas nécessairement beaucoup de points et qui ont pour but de vérifier que vous avez compris ;
- Il n'y avait pas de questions particulièrement difficile une fois que les structures étaient comprises, mais cela pourrait ne pas être le cas dans la suite.

Remarques sur la notation

- Dans le cas où il y avait une erreur dans le sujet je vous ai accordé les points si vous avez fait quelque chose de cohérent avec le reste.

Caractère, chaînes de caractère

- Attention à la manipulation des chaînes de caractères : depuis la fin de Caml light, les chaînes sont non mutable : vous ne pouvez pas les modifier ;
- Attention à int_of_char, cela ne fait pas ce que vous pensez que cela fait :

```
1 int_of_char '1'
```

```
49
```

- Vous ne pouvez pas concaténer des caractères comme vous le feriez avec des chaînes de caractères, il fallait utiliser des choses plus élémentaires.

Syntaxe

- Toujours ces points virgules qui sont en trop ;
- Quelques soucis au cas par cas.

Organisation du code

- Mettez vos fonctions auxilaires au début de la fonction, pas à l'intérieur d'une boucle `for` si possible ;
- Indentation et sauts de lignes simplifient la correction.

Nommer les arguments

```
let f t =
    let n = Array.length t in
    for i = 0 to n - 1 do
        let rec aux t.(i) = ...
        in ...
    done
```

Nommer les arguments

```
let f t =
    let n = Array.length t in
    for i = 0 to n - 1 do
        let rec aux t.(i) = ...
        in ...
    done
```

```
    let f t =
    let n = Array.length t in
    for i = 0 to n - 1 do
        let rec aux l = ...
        in aux t.(i)
    done
```

Nommer les arguments

```
let f t =
    let n = Array.length t in
    for i = 0 to n - 1 do
        let rec aux t.(i) = ...
        in ...
    done
```

```
    let f t =
    let n = Array.length t in
    for i = 0 to n - 1 do
        let rec aux l = ...
        in aux t.(i)
    done
```

```
    let f t =
    let n = Array.length t in
    let rec aux l = ...
    in
    for i = 0 to n - 1 do
        aux t.(i)
    done
```

Typage

Attention au typage des fonctions que vous utilisez : elles doivent avoir le bon type.

```
1 let diviser (a,b) (c, d) =
2     (multiplier (a,b) (conjuguer (c,d)))/. (
      multiplier (c,d) (conjuguer (c,d)))
```

Typage

Attention au typage des fonctions que vous utilisez : elles doivent avoir le bon type.

```
1 let diviser (a,b) (c, d) =
2     (multiplier (a,b) (conjuguer (c,d)))/. (
    multiplier (c,d) (conjuguer (c,d)))
```

```
1 let diviser (a,b) (c, d) =
2     let denominateur = fst ((multiplier (c,d) (
    conjuguer (c,d))) in
3     (multiplier (a,b) (conjuguer (c,d)))/.
    denominateur)
```

Mutabilité

Attention, encore trop de codes qui ne font *rien*, vous devez :

- renvoyer quelque chose quand demandé, et `unit` sinon ;
- modifier les choses que vous voulez renvoyer si vous les construisez au fur et à mesure.

```
let convertir t =
    let n = Array.length l in
    let l = [] in
    for i = 0 to n-1 do
        t.(n-i-1)::l
    done ; l
```

Mutabilité

Attention, encore trop de codes qui ne font *rien*, vous devez :

- renvoyer quelque chose quand demandé, et `unit` sinon ;
- modifier les choses que vous voulez renvoyer si vous les construisez au fur et à mesure.

```
let convertir t =
    let n = Array.length l in
    let l = [] in
    for i = 0 to n-1 do
        t.(n-i-1)::l
    done ; l
```

Par ailleurs, les chaînes de caractère ne sont pas mutables : vous ne pouvez pas les modifier :

```
"aui".[1] <- 'c'
```

Code un peu alambiqué

Beaucoup de vos codes sont inutilements compliqués à cause des références :

```
let echanger a b =
    let t = ref !a in
    a:=!b;
    b:=!t
```

Code un peu alambiqué

Beaucoup de vos codes sont inutilements compliqués à cause des références :

```
let echanger a b =
    let t = ref !a in
    a:=!b;
    b:=!t
```

```
let echanger a b =
    let t = a in
    a:=!b;
    b:=!t
```

Code un peu alambiqué

Beaucoup de vos codes sont inutilements compliqués à cause des références :

```
let echanger a b =
    let t = ref !a in
    a:=!b;
    b:=!t
```

```
let echanger a b =
    let t = a in
    a:=!b;
    b:=!t
```

```
let echanger a b =
    let t = !a in
    a:=!b;
    b:=t
```

Utilisation de la programmation impérative

En règle générale, vous utilisez beaucoup de références. Cela peut être un souci si :

- vous utilisez des références qui seraient plus simple à manipuler avec des arguments d'une fonction auxiliaire (par exemple des références de listes, ou bien des références alors que vous avez défini une fonction auxiliaire qui manipule cette référence pour y ajouter le résultat) ;
- vous utilisez des références par défaut parce que vous ne savez pas faire autrement.

Table de hachage

- On cherche l'élément dans une case précise, pareil pour retirer l'élément ;
- La complexité est en $O(n + N)$ pour la plupart des opérations qui opèrent sur tous les éléments de la table.

Construire son propre type en OCaml

Définition de type

On peut définir un nouveau type avec la syntaxe `type ... = ...`. Cela permet par exemple de définir des types à partir d'autres types :

```
type polynome = int list
```

```
type point = float * float
```

Type somme : booléen

Un type somme est un type qui est une union de plusieurs possibilités.
On peut par exemple définir le type booleen de la manière suivante :

```
1 type booleen = Vrai | Faux
```

Vrai et Faux sont appelés constructeurs, et doivent commencer par une majuscule.

Opérations sur le type Booléen

```
let ou x y = match x, y with
| Vrai, _ -> Vrai
| _, Vrai -> Vrai
| _ -> Faux
```

Opérations sur le type Booléen

```
let ou x y = match x, y with
| Vrai, _ -> Vrai
| _, Vrai -> Vrai
| _ -> Faux
```

Proposer une fonction et similaire.

Opérations sur le type Booléen

```
let ou x y = match x, y with
| Vrai, _ -> Vrai
| _, Vrai -> Vrai
| _ -> Faux
```

Proposer une fonction et similaire.

```
let et x y = match x, y with
| Vrai, Vrai -> Vrai
| _ -> Faux
```

Mot-clef of : droite réelle achevée (1)

$$\overline{\mathbb{R}} = \{-\infty, +\infty\} \cup \mathbb{R}$$

```
type r_barre = PlusInf | MoinsInf | Reel of float
```

Mot-clef of : droite réelle achevée (1)

$$\overline{\mathbb{R}} = \{-\infty, +\infty\} \cup \mathbb{R}$$

```
type r_barre = PlusInf | MoinsInf | Reel of float
```

```
let est_strictement_plus_grand r1 r2 = match r1, r2
    with
| MoinsInf, _ -> false
| _, PlusInf -> false
| PlusInf, _ -> true
| _, MoinsInf -> true
| Reel x1, Reel x2 -> x1>x2
```

Mot-clef of : droite réelle achevée (1)

$$\overline{\mathbb{R}} = \{-\infty, +\infty\} \cup \mathbb{R}$$

```
type r_barre = PlusInf | MoinsInf | Reel of float
```

```
let est_strictement_plus_grand r1 r2 = match r1, r2
    with
| MoinsInf, _ -> false
| _, PlusInf -> false
| PlusInf, _ -> true
| _, MoinsInf -> true
| Reel x1, Reel x2 -> x1>x2
```

En déduire une fonction qui renvoie le maximum de deux arguments :

```
let maximum r1 r2 =
    if est_strictement_plus_grand r1 r2
    then r1
    else r2
```

Mot-clef of : droite réelle achevée (2)

$$\overline{\mathbb{R}} = \{-\infty, +\infty\} \cup \mathbb{R}$$

```
type r_barre = PlusInf | MoinsInf | Reel of float
```

Multiplication ?

Mot-clef of : droite réelle achevée (2)

$$\overline{\mathbb{R}} = \{-\infty, +\infty\} \cup \mathbb{R}$$

```
type r_barre = PlusInf | MoinsInf | Reel of float
```

Multiplication ?

```
let rec mult r1 r2 =
    if est_strictement_plus_grand r2 r1 then mult r2
     r1
    else match r1, r2 with
    | PlusInf, PlusInf | MoinsInf, _ -> PlusInf
    | PlusInf, MoinsInf -> MoinsInf
    | PlusInf, Reel x1 when x1>0. | Reel x1 ,
    MoinsInf when x1 < 0.-> PlusInf
    | PlusInf, Reel x1 when x1<0. | Reel x1,
    MoinsInf when x1 > 0. -> MoinsInf
    | Reel x1, Reel x2 -> Reel (x1 *. x2)
    | _ -> failwith "Non defini"
```

Type paramétrique : option

Un type peut être défini comme paramétrique :

```
1 type 'a option = None | Some of 'a
```

Type paramétrique : option

Un type peut être défini comme paramétrique :

```
type 'a option = None | Some of 'a
```

```
let rec dernier l = match l with
| [] -> None
| [v] -> Some v
| _::q -> dernier q
```

Type paramétrique : option

Un type peut être défini comme paramétrique :

```
type 'a option = None | Some of 'a
```

```
let rec dernier l = match l with
| [] -> None
| [v] -> Some v
| _::q -> dernier q
```

```
let extraire valeur default = match valeur with
| Some v -> v
| None -> default
```

Type paramétrique : option

Un type peut être défini comme paramétrique :

```
type 'a option = None | Some of 'a
```

```
let rec dernier l = match l with
| [] -> None
| [v] -> Some v
| _::q -> dernier q
```

```
let extraire valeur default = match valeur with
| Some v -> v
| None -> default
```

Le type `'a option` est défini par défaut en OCaml : vous pouvez l'utiliser sans le définir.

Type récursif : Liste chaînée

Un type peut être lui-même récursif :

```
1 type 'a liste = Vide | Suite of 'a * 'a liste
```

Type récursif : Liste chaînée

Un type peut être lui-même récursif :

```
type 'a liste = Vide | Suite of 'a * 'a liste
```

On peut réaliser des calculs sur ce type récursif comme s'il s'agissait de listes :

```
let rec longueur l = match l with
| Vide -> 0
| Suite (_, q) -> 1 + longueur q
```

Type récursif : Liste chaînée

Un type peut être lui-même récursif :

```
type 'a liste = Vide | Suite of 'a * 'a liste
```

On peut réaliser des calculs sur ce type récursif comme s'il s'agissait de listes :

```
let rec longueur l = match l with
| Vide -> 0
| Suite (_, q) -> 1 + longueur q
```

Proposer une fonction ajouter qui ajoute un élément à la fin d'une liste.

Type récursif : Liste chaînée

Un type peut être lui-même récursif :

```
type 'a liste = Vide | Suite of 'a * 'a liste
```

On peut réaliser des calculs sur ce type récursif comme s'il s'agissait de listes :

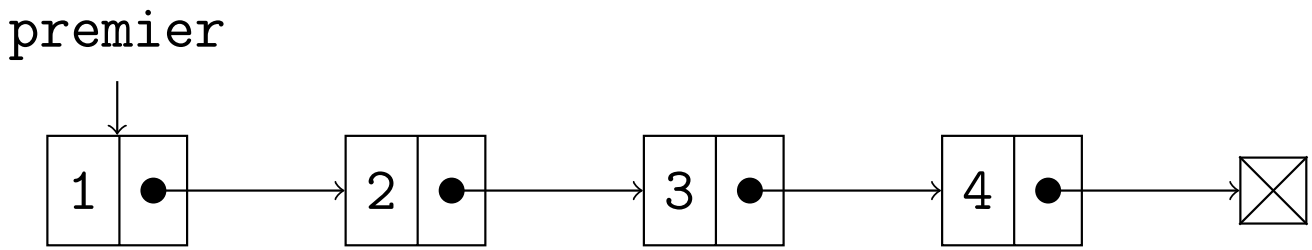
```
let rec longueur l = match l with
| Vide -> 0
| Suite (_, q) -> 1 + longueur q
```

Proposer une fonction ajouter qui ajoute un élément à la fin d'une liste.

```
let rec ajouter l x = match l with
| Vide -> Suite (x, Vide)
| Suite(p, q) -> Suite(p, ajouter q x)
```

Liste chaînée

En réalité, le type que l'ont vient de définir correspond exactement aux listes, et il s'avère que cela correspond à un type que l'on croise souvent : les listes chaînées.



Les opérations sur les listes chaînées sont donc les mêmes que sur les listes OCaml. Mais nous verrons d'autres types de listes chaînées en OCaml.

Type produit : complexe

On peut définir un type **produit** (ou **enregistrement**) avec la syntaxe :

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

Type produit : complexe

On peut définir un type **produit** (ou **enregistrement**) avec la syntaxe :

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

```
let somme c1 c2 =
    let {re=x1; im=y1} = c1 in
    let x2 = c2.re in
    let y2 = c2.im in
    {re = x1 +. y1; im = y1 +. y2}
```

Enregistrement mutable

On peut créer des enregistrement mutables, c'est-à-dire faire des champs où les données peuvent être modifiées par effet de bord.

```
type mut = {mutable a : int; b : int}
let x = {a = 0; b = 1}
let () = x.a <- 1
```

```
x.a
```

Enregistrement mutable

On peut créer des enregistrement mutables, c'est-à-dire faire des champs où les données peuvent être modifiées par effet de bord.

```
type mut = {mutable a : int; b : int}
let x = {a = 0; b = 1}
let () = x.a <- 1
```

```
x.a
```

```
1
```

Liste chaînée mutable (1)

On peut créer un type de liste chaînée mutable avec le type suivant :

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Liste chaînée mutable (1)

On peut créer un type de liste chaînée mutable avec le type suivant :

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment peut-on convertir notre liste vers une liste classique d'OCaml ?

Liste chaînée mutable (1)

On peut créer un type de liste chaînée mutable avec le type suivant :

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment peut-on convertir notre liste vers une liste classique d'OCaml ?

```
let vers_persistant lc =
    let rec aux l = match l with
        | Vide -> []
        | C(c) -> (c.valeur)::(aux c.suiv)
    in aux lc.premier
```

Liste chaînée mutable (2)

On peut créer un type de liste chaînée mutable avec le type suivant :

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment faire la conversion inverse ?

Liste chaînée mutable (2)

On peut créer un type de liste chaînée mutable avec le type suivant :

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment faire la conversion inverse ?

```
let vers_mutable l =
    let rec aux l = match l with
        | [] -> Vide
        | p::q -> let qmut = aux q in
        Cell({valeur= p; suiv = qmut})
    in
    {premier = aux l}
```

Liste chaînée mutable (3)

```
1 type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
2 type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment rajouter un élément en début de liste avec une fonction de signature `'a list_chainee -> 'a -> unit` ?

Liste chaînée mutable (3)

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment rajouter un élément en début de liste avec une fonction de signature `'a list_chainee -> 'a -> unit` ?

```
let rajouter lc x =
    let nouvelle = {valeur= x; suiv = lc.premier} in
    lc.premier <- nouvelle
```

Liste chaîne mutable (4)

```
1 type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
2 type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment retirer un élément en fin de liste ?

Liste chaîne mutable (4)

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Comment retirer un élément en fin de liste ?

```
let retirer lc = match lc.premier with
| Vide -> failwith "Liste vide"
| Cell({valeur = _ ; suiv = Vide}) -> lc.premier <-
    Vide
| _ ->
    let rec aux l = match l with
    | Vide -> failwith "Liste vide"
    | Cell({valeur = _ ; suiv = Vide}) -> failwith "
    Liste reduite a un element"
    | Cell({valeur = _ ; suiv = Cell(c)}) ->
        if c.suiv = Vide
            then l.suiv <- Vide
            else aux c.suiv
    in aux lc.premier
```