Algorithmique - Programmation 1

Cours 7

Université Henri Poincaré

CESS Epinal

Automne 2008

Plan

Rappel : le produit cartésien

Enregistrements (produits nommés)

Termes

Application : dimension de formules physiques

Rappel : le produit cartésien

- Exemples d'utilisation :
  - modélisation des points du plan $(x, y) \in \mathbb{R} \times \mathbb{R}$
    $\longrightarrow$ `float * float`
  - modélisation des nombres complexes $z = a + ib \in \mathbb{C}$
    $\longrightarrow$ `float * float`
- Problème : comment vérifier que les données manipulées sont compatibles ?

Plan

Rappel : le produit cartésien

Enregistrements (produits nommés)

Termes

Application : dimension de formules physiques

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
```

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
type complexe = {preelle :  float; pimg :  float;}
```

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
type complexe = {preelle :  float; pimg :  float;}
```

- NB: les dimensions (appelées aussi champs) sont séparés par des **point-virgules**

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
type complexe = {preelle :  float; pimg :  float;}
```

- NB: les dimensions (appelées aussi champs) sont séparés par des **point-virgules**
- Utilisation (**définition + accès en lecture**) :

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
type complexe = {preelle :  float; pimg :  float;}
```

- NB: les dimensions (appelées aussi champs) sont séparés par des **point-virgules**
- Utilisation (**définition + accès en lecture**) :

```
# let z = {preelle = 1.5 ; pimg = 2.3};;
val z :  complexe = {preelle = 1.5; pimg = 2.3}
```

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
type complexe = {preelle :  float; pimg :  float;}
```

- NB: les dimensions (appelées aussi champs) sont séparés par des **point-virgules**
- Utilisation (**définition + accès en lecture**) :

```
# let z = {preelle = 1.5 ; pimg = 2.3};;
val z :  complexe = {preelle = 1.5; pimg = 2.3}

# let norme = function x -> sqrt (x.preelle ** 2.
                                  +.  x.pimg ** 2.);;
```

Enregistrements (produits nommés)

- Un enregistrement est un produit cartésien dans lequel **chaque dimension est nommée** (en plus d'être typée)

  Exemple :

```
# type complexe = {preelle :float; pimg :float};;
type complexe = {preelle :  float; pimg :  float;}
```

- NB: les dimensions (appelées aussi champs) sont séparés par des **point-virgules**
- Utilisation (**définition + accès en lecture**) :

```
# let z = {preelle = 1.5 ; pimg = 2.3};;
val z :  complexe = {preelle = 1.5; pimg = 2.3}
```

```
# let norme = function x -> sqrt (x.preelle ** 2.
                                  +.  x.pimg ** 2.);;
val norme :  complexe -> float = <fun>
```

Plan

Rappel : le produit cartésien

Enregistrements (produits nommés)

Termes

Application : dimension de formules physiques

Termes

- Un terme est un nouveau type de données construit en utilisant des *étiquettes*, exemples :

```
# type couleur = Vert | Rouge;;
type couleur = Vert | Rouge

# type entier = Entier of int;;
type entier = Entier of int
```

- L'étiquette est appelée **constructeur** de type, elle commence par une majuscule et peut être paramétrée
- Exemple d'utilisation :

```
# let somme (Entier a) (Entier b) = Entier (a+b);;
val somme :  entier -> entier -> entier = <fun>
```

- Les termes permettent :
  1. de regrouper plusieurs types en un seul (**types sommes**)
  2. de représenter des types complexes (**types récursifs**)

Plan

Rappel : le produit cartésien

Enregistrements (produits nommés)

Termes

- Types sommes
- Types récursifs

Application : dimension de formules physiques

Types sommes

- Un type somme (ou **énumération**) permet de regrouper plusieurs types de données
- Une énumération utilise la **barre verticale** (|) comme séparateur de type, exemple :

```
# type nombre =
        Entier of int
      | Reel   of float;;
type nombre = Entier of int | Reel of float
```

- NB : les étiquettes **commencent par une majuscule** !
- **Règle de typage** : soit le type `t` défini par

```
# type t = Etiquette1 of t1|...|EtiquetteN of tN;;
```

alors une expression de la forme (`EtiquetteI x`) est de type `t` si `x` est de type `tI`

Types sommes (suite)

- Pour définir un traitement sur un type somme, on utilise généralement le filtrage (*pattern matching*) sur les étiquettes :

```
# let float_of_nombre =
                function Entier n -> float_of_int n
                       | Reel n   -> n;;
val float_of_nombre : nombre -> float = <fun>

# let somme x y = match x,y with
    (Entier a), (Entier b) -> Entier (a+b)
  | (Entier a), (Reel b)   -> Reel ((float_of_int a)+.b)
  | (Reel a)  , (Entier b) -> Reel (a+.(float_of_int b))
  | (Reel a)  , (Reel b)   -> Reel (a+.b);;
val somme : nombre -> nombre -> nombre = <fun>
```

- NB : il y a autant de cas qu'il y a d'étiquettes !

Types sommes (suite)

- Pour retourner une donnée de type somme, il suffit de lui associer la bonne étiquette :

```
# let nombre_of_int = function x -> Entier x;;
val nombre_of_int : int -> nombre = <fun>

# let nombre_of_float = function x -> Reel x;;
val nombre_of_float : float -> nombre = <fun>
```

- Il est possible de combiner les concepts de liste et de type somme :

```
let rec sommeAll = function
                []    -> Entier 0
             | t::r -> somme t (sommeAll r);;
val sommeAll : nombre list -> nombre = <fun>
```

- Comment réécrire la fonction `sommeAll` au moyen de `List.fold_left` ?

Plan

Rappel : le produit cartésien

Enregistrements (produits nommés)

Termes
- Types sommes
- Types récursifs

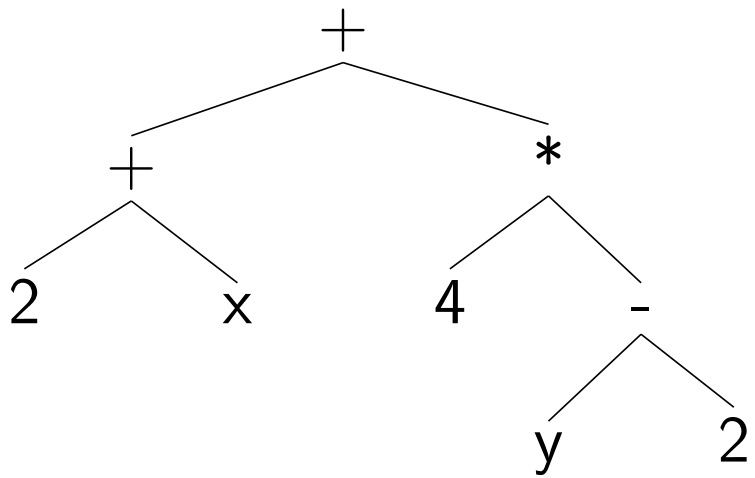
Application : dimension de formules physiques

Types récursifs

- Les types récursifs permettent de représenter des données complexes
- Exemple : les expressions mathématiques

$$2 + x + 4 \times (y - 2)$$

que l'on peut représenter sous forme d'arbre :



Types récursifs (suite)

- Une formule est composée :

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules
  - **ou** d'une différence de sous-formules

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules
  - **ou** d'une différence de sous-formules
  - **ou** d'une multiplication de sous-formules

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules
  - **ou** d'une différence de sous-formules
  - **ou** d'une multiplication de sous-formules
  - **ou** d'une division de sous-formules

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules
  - **ou** d'une différence de sous-formules
  - **ou** d'une multiplication de sous-formules
  - **ou** d'une division de sous-formules
  - **ou** d'une constante (entière pour simplifier)

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules
  - **ou** d'une différence de sous-formules
  - **ou** d'une multiplication de sous-formules
  - **ou** d'une division de sous-formules
  - **ou** d'une constante (entière pour simplifier)
  - **ou** d'une variable

Types récursifs (suite)

- Une formule est composée :
  - d'une somme de sous-formules
  - **ou** d'une différence de sous-formules
  - **ou** d'une multiplication de sous-formules
  - **ou** d'une division de sous-formules
  - **ou** d'une constante (entière pour simplifier)
  - **ou** d'une variable
- Ce que l'on peut représenter via le type récursif `formule` suivant :

```
# type formule = Plus of formule * formule
               | Moins of formule * formule
               | Fois of formule * formule
               | Sur of formule * formule
               | Coef of int
               | Var of string ;;
```

Types récursifs (suite)

- Ainsi, l'expression

$$2 + x + 4 \times (y - 2)$$

peut s'écrire via :

```
# let e = Plus (Plus (Coef 2, Var "x"),
                  Fois (Coef 4,
                        Moins (Var "y", Coef 2)));;

val e : formule =
Plus (Plus (Coef 2, Var "x"),
      Fois (Coef 4, Moins (Var "y", Coef 2)))
```

Les termes (suite)

- Quelques points à noter :
  - Les termes peuvent être définis récursivement
  - Les étiquettes s'appliquent à des types de données complexes ou non (en particulier sur des produits cartésiens, *cf* type `formule`)
  - Il est possible de modéliser des données plus ou moins complexes
  - Certaines données se modélisent "naturellement" de manière récursive (*cf* exemple slides suivants)

Plan

Rappel : le produit cartésien

Enregistrements (produits nommés)

Termes
- Types sommes
- Types récursifs

Application : dimension de formules physiques

Application : dimension de formules physiques

- Problème : vérifier la dimension des données manipulées au sein de formules physiques

  Exemple :

$$\Delta_d = v_0\Delta_t + \frac{a{\Delta_t}^2}{2} \quad \checkmark$$

$$\Delta_d = v_0\Delta_t + \frac{a\Delta_t}{2} \quad \times$$

- Démarche :

  1. Modélisation du problème (type “dimension”)
  2. Représentation des données (formules physiques)
  3. Calcul des dimensions

Application : dimension de formules physiques

- 1. Dimension d'une mesure physique représentée en fonction de **dimensions de base** :
  - longueur
  - masse
  - temps

Application : dimension de formules physiques

- 1. Dimension d'une mesure physique représentée en fonction de **dimensions de base** :
  - longueur
  - masse
  - temps
- **Utilisation d'un vecteur** dont les composantes sont les **dimensions de base** :

$$\vec{d} = (\underset{\text{longueur}}{x}, \underset{\text{masse}}{y}, \underset{\text{temps}}{z})$$

Application : dimension de formules physiques

- 1. Dimension d'une mesure physique représentée en fonction de **dimensions de base** :
  - longueur
  - masse
  - temps
- **Utilisation d'un vecteur** dont les composantes sont les **dimensions de base** :

$$\vec{d} = (\underbrace{x}_{\text{longueur}}, \underbrace{y}_{\text{masse}}, \underbrace{z}_{\text{temps}})$$

- En `Caml` :

```
# type dimension = int * int * int;;
```

Application : dimension de formules physiques

- Exemples de dimensions :

$$\text{vitesse } v \quad \longrightarrow \quad \vec{d_v} = (1, 0, -1) \quad cf\,(m/s)$$

$$\text{gravité } g \quad \longrightarrow \quad \vec{d_g} = (1, 0, -2) \quad cf\,(m/s^2)$$

- 2. Une fois les dimensions modélisées, comment représenter une formule physique ?
  $\longrightarrow$ Utilisation d'un **type récursif** `formule`

- En `Caml` :

```
# type formule = Plus of formule * formule
               | Moins    of formule * formule
               | Fois     of formule * formule
               | Sur      of formule * formule
               | Expo     of formule * int
               | Coef     of float
               | Var      of string ;;
```

Application : dimension de formules physiques

- 3. Calcul des dimensions :

  $\longrightarrow$ fonction prenant en entrée une formule physique et retournant une dimension (*i.e.* un vecteur)

- Règles de combinaison des dimensions :
  - Multiplication de sous-formules
    $\longrightarrow$ addition des dimensions $(x_1, y_1, z_1) + (x_2, y_2, z_2)$
  - Division de sous-formules
    $\longrightarrow$ soustraction $(x_1, y_1, z_1) - (x_2, y_2, z_2)$
  - Exposant d'une sous-formule
    $\longrightarrow$ multiplication par un scalaire $c \times (x_1, y_1, z_1)$
  - Addition / soustraction de sous-formules
    $\longrightarrow$ comparaison $(x_1, y_1, z_1) = (x_2, y_2, z_2)$

Application : dimension de formules physiques

- Calcul des dimensions au moyen d'une fonction récursives, parcourant la formule passée en entrée :

```
# let rec dimen = function
      Fois(a1,a2) -> add_dim (dimen a1) (dimen a2)
    | Sur(a1,a2)  -> sub_dim (dimen a1) (dimen a2)
    | Expo(a1,n)  -> mult_dim (dimen a1) n
    | Plus(a1,a2) -> if  (dimen a1) = (dimen a2)
                     then dimen a1
                     else failwith "erreur sur +"
    | Moins(a1,a2)-> if (dimen a1) = (dimen a2)
                     then dimen a1
                     else failwith "erreur sur -"
    | Coef _      -> (0, 0, 0)
    | Var vp      -> dim_var vp ;;

val dimen : formule -> int * int * int = <fun>
```

Application : dimension de formules physiques

- Attention :
  - les fonctions `dimen_var`, `add_dim`, `sub_dim` et `mult_dim` doivent être définies

Application : dimension de formules physiques

- Attention :
  - les fonctions `dimen_var`, `add_dim`, `sub_dim` et `mult_dim` doivent être définies
- Dimension des variables physiques :

```
let dim_var = function
     "gamma" -> (1, 0, -2)
   | "l" -> (1, 0, 0)
   | "m" -> (0, 1, 0)
   | "t" -> (0, 0, 1)
   | x -> failwith ("var inconnue :"^ x);;
```

Application : dimension de formules physiques

- Attention :
  - les fonctions `dimen_var`, `add_dim`, `sub_dim` et `mult_dim` doivent être définies
- Dimension des variables physiques :

```
let dim_var = function
     "gamma" -> (1, 0, -2)
    | "l" -> (1, 0, 0)
    | "m" -> (0, 1, 0)
    | "t" -> (0, 0, 1)
    | x -> failwith ("var inconnue :"^ x);;
```

- Addition / soustraction / multiplication par un scalaire :

```
# let add_dim (x1,y1,z1) (x2,y2,z2) =
(x1+x2,y1+y2,z1+z2);;
```