Plan des 4 prochains cours

- Principes du typage statique
- Types de données

Les principes du typage

Plan

- Introduction au typage
- Quelques exemples
- Notations
- Le typage de CAML
- Modularité
- traits impératifs
- L'unification

Introduction I

Le typage Diviser l'espace des valeurs en collections: un type est une collection de valeurs partageant une même propriété

Les types sont toujours plus ou moins vérifiés:

- soit à l'exécution (typage dynamique)
- soit à la compilation (typage statique)

Intérêts du typage statique

- détection précoce d'erreurs
- documentation, structuration (une première information sur le programme)
- facilite les optimisations (enregistrements, accès, représentations,...)
- élimination de certaines erreurs d'exécution
- sécurité du code

Introduction II

Remarque

- le typage dynamique est plus précis (`if B then 1 else 2+"a"`, avec `B` = vrai)
- le typage statique doit être conservatif
- trouver des systèmes de type les plus riches possible

On étudie ici les principes du typage statique

Qu'est qu'un type?

- Un type est une expression d'un langage de types
- Déclarer un type = introduire un nom de type
- Définir un type = associer un nom à une exp. de type

Typer une expression

- Typer = associer un type à une expression
- Typage fort = toute expression a un type et un seul

Introduction III

Vérification des types le programmeur associe un type à chaque déclaration d'identificateur et le compilateur vérifie la correction (e.g, C, C++, Pascal, Ada,...)

```
int addition (int x1, int x2)
{  int temp;
   temp = x1 + x2;
   return(temp); }
```

Synthèse des types les types sont devinés (on dira synthétisés ou inférés) par le compilateur par une analyse des contraintes d'utilisation des variables (e.g, Caml, SML,...)

```
#let addition x1 x2 =
#  let temp = x1 + x2 in
#  temp;;
```

Introduction IV

"le programme P est de type ty"

$$P : ty$$

Vérification

- vérifier que ty est un type valide pour P

Synthèse

- trouver un ty valide

Intérêts de la synthèse

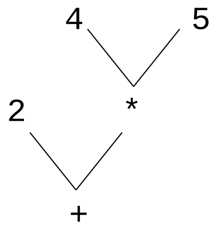
- concision, simplicité, facilité d'écriture
- généralité plus grande des programmes
- Un algorithme fondamental de l'informatique (IA)

Quelques exemples I

Typer une expression: parcourir l'expression en associant un type unique à chaque sous-expression

Exemple: `2 + (4 * 5)`

$$\dfrac{2 : \texttt{int} \qquad \dfrac{4 : \texttt{int} \quad 5 : \texttt{int}}{4 * 5 : \texttt{int}}}{2 + (4 * 5) : \texttt{int}}$$



Un axiome:

$$i : \texttt{int}$$

Une règle de déduction:

$$\dfrac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}}$$

Quelques exemples II

- des axiomes:

$$P : ty$$

- des règles d'inférence

$$\frac{P_1 : ty_1 \quad P_n : ty_n}{C(P_1, ..., P_n) : ty}$$

- typer = construire un arbre où les feuilles sont des axiomes et les noeuds, des règles d'inférence

$$\frac{\begin{array}{cc} \vdots & \vdots \\ P_1 : ty_1 & P_n : ty_n \end{array}}{C(P_1, ..., P_n) : ty}$$

Quelques exemples III

Un programme est bien typé lorsque l'on peut construire une telle déduction

Typer: `2 + (true * 5)`

$$\frac{2 : \mathtt{int} \quad \dfrac{\mathtt{true} : \mathtt{bool} \quad 5 : \mathtt{int}}{\mathtt{true} * 5 : ?}}{2 + (\mathtt{true} * 5) : ?}$$

Question: Que faire en présence d'identificateurs?

```
#let x = 1;;

#let y = 2 + (4 * x);;
```

On introduit l'environnement de typage.

- contient des liaisons (*idf*, *type*)
- $\lfloor(\mathtt{y} : \mathtt{int})(\mathtt{x} : \mathtt{int})\rfloor = Env$

Notations, définitions I

"Dans l'environnement *Env*, le programme P a le type *ty*"

$$Env \vdash P : ty$$

L'environnement de typage

- liste d'association $\lfloor (x_1 : ty_1), ..., (x_n : ty_n) \rfloor$
- une fonction d'accès *acces* tq:

$$acces\ (x, \lfloor (x_1 : ty_1), ..., (x_n : ty_n) \rfloor) = \begin{cases} ty_1 \text{ si } x = x_1 \\ acces\ (x, \lfloor (x_2 : ty_2), ..., (x_n : ty_n) \rfloor) \text{ sinon} \end{cases}$$

Notre langage des types

- *Types de base*: `unit`, `int`, `float`, `char`, `string`,...
- *Types construits*
  - types constants déclarés
  - constructeur de type (opérateur sur les types). e.g, `*`
  - types paramétrés, e.g, `('a,'b) arbre`
- *Variables de type* :α, β,... notés `'a,'b,...`

Le typage de CAML I

Constantes

$$Env \vdash \texttt{true} : \texttt{bool}$$

$$Env \vdash \texttt{false} : \texttt{bool}$$

$$Env \vdash \texttt{42} : \texttt{int}$$

Identificateurs

$$(\textsc{Ident})\ \frac{acces\ (x, Env) = ty}{Env \vdash x : ty}$$

Conditionnelle

`if` *cond* `then` *e1* `else` *e2*

$$(\textsc{Cond})\ \frac{Env \vdash cond : bool \quad Env \vdash e1 : ty \quad Env \vdash e2 : ty}{Env \vdash \texttt{if}\ cond\ \texttt{then}\ e1\ \texttt{else}\ e2 : ty}$$

Le typage de CAML II

Application de fonction

Soient e_1 une expression (fonctionnelle) et e_2 une expression. L'application $e_1\ e_2$ est typée avec la règle :

$$(\textsc{App})\ \frac{Env \vdash\ e_1\ :\ t_1 \rightarrow t_2 \quad Env \vdash\ e_2\ :\ t_1}{Env \vdash\ e_1\ e_2\ :\ t_2}$$

Exemples:

`fact (fact 5)`

`2 + (4 * 5)`

Remarque:

- `->` est associatif à droite
  (t_1 `->` t_2 `->` t_3 `=` t_1 `->` `(`t_2 `->` t_3`)`)
- l'application est associative à gauche
  $e_1\ e_2\ e_3 = (e_1\ e_2)\ e_3$

Le typage de CAML III

Fonctions: `function` x `->` *corps*

Typage dans *Env*

1. Choisir une nouvelle variable de type, τ, pour désigner le type de x. τ est donc une inconnue.
2. Typer *corps* dans $\lfloor (x, \tau) \triangleleft\ Env \rfloor$. Fait apparaître des contraintes de typage ($\tau = t_1$).
3. Faire le bilan des contraintes :
   (a) contraintes contradictoires : erreur de typage.
   (b) variables de type toutes déterminées: si $x : t_1$ et si `corps`: t_2, alors `function` x `->` *corps*: $t_1 \rightarrow t_2$
   (c) certaines variables de type restent indéterminées: polymorphisme (plus tard)

$$(\textsc{Fun})\ \frac{\lfloor (x : t_1) \triangleleft\ Env \rfloor\ \vdash\ corps\ :\ t_2}{Env \vdash\ \texttt{function}\ x\ \texttt{->}\ corps\ :\ t_1 \rightarrow t_2}$$

Le typage de CAML IV

Déclarations globales

```
let idf = exp;;
```

```
#let x = 1;;
#let y = x+x;;
#let y = y+2;;
```

1. Typage dans l'env. global courant Env
2. typage de exp

$$Env \vdash\ exp\ :\ ty$$

3. le nouvel environnement global
   $Env' = \lfloor (idf : ty) \triangleleft\ Env \rfloor$

Le typage de CAML V

Déclarations locales

```
let idf = exp1 in exp2;;
```

```
#let z = 1 in z+2;;
#let y = z+1;;
```

1. Typage de $exp1$ ($Env \vdash\ exp1\ :\ ty_1$)
2. Ajout du type obtenu à l'environnement
   $Env\_loc = \lfloor (idf : ty_1) \triangleleft\ Env \rfloor$
3. Typage de $exp2$ dans Env_loc
   ($Env' \vdash\ exp2\ :\ ty_2$)
4. L'environnement global est inchangé

$$(\textsc{Let})\ \frac{Env \vdash\ exp_1\ :\ t_1 \quad \lfloor (idf : t_1) \triangleleft\ Env \rfloor \vdash\ exp_2\ :\ t_2}{Env \vdash\ \texttt{let}\ idf\ \texttt{=}\ exp_1\ \texttt{in}\ exp_2\ :\ t_2}$$

Le typage de CAML VI

Fonctions récursives une fonction récursive est une expression fonctionnelle déclarée par `let rec`.

let rec f = function x -> *corps*

```
#let rec fact1 = function n ->
#    if n <= 0 then 1
#    else n * fact1 (n-1);;
```

Typage dans *Env*

1. Choisir deux nouvelles variables de type τ_1 et τ_2 pour désigner le type de `f` puis étendre *Env*
   $Env' = \lfloor (f : \tau_1 \rightarrow \tau_2), (x : \tau_1) \triangleleft\ Env \rfloor$
2. typer *corps* ($Env' \ \vdash \ corps \ : \ ty_2$)
3. Faire le bilan des contraintes ($\tau_1 = t_1$ et $\tau_2 = t_2$)

$$(\text{Rec})\frac{\lfloor (f : t_1 \rightarrow t_2), (x : t_1) \triangleleft\ Env \rfloor \ \vdash \ corps \ : \ t_2}{Env \vdash \texttt{rec}\ f\ \texttt{= function}\ x\ \texttt{->}\ corps : t_1 \rightarrow t_2}$$

Le typage de CAML VII

Polymorphisme Lors du typage, certaines variables de type restent indéterminées. Le programme est très général !

```
#let app_tab f t =
#  for i = 0 to vect_length t - 1 do
#    t.(i) <- f (t.(i))
#  done;;
```

```
#let ajout n t =
#  app_tab (function x -> x+n) t;;
```

```
#let not_tab t =
#  app_tab (function x -> not x) t;;
```

Intérêt du polymorphisme

- généralité du code, concision
- réutilisabilité (bibliothèques)

Le typage de CAML VIII

Les variables de type non déterminées sont généralisées.

$$Env \vdash \texttt{function x -> x} : \forall\alpha.\alpha \to \alpha$$

Schéma de type: expression de type dont certaines variables, notées `'a`, sont quantifiées universellement en tête de l'expression: $\forall\alpha.\alpha \to \alpha$. Les variables non quantifiées, notées `_'a`, sont dites *libres*.

Type: expression de type sans quantificateur.

Type monomorphe: type ne contenant pas de variable de type

Type polymorphe: schéma de type

Instancier un schéma de type: remplacer les variables universellement quantifiées par une expression de type.

Généraliser un type dans *Env*: quantifier universellement les variables du type qui ne sont pas libres dans *Env*.

Le typage de CAML IX

Typage d'une expression ne contenant que des identificateurs connus

Cas monomorphe: déjà vu.

Cas polymorphe: $\forall\alpha_1 \dots \alpha_n.typ$
instancier chaque α_i par une *nouvelle* variable de type τ_i.

On modifie la règle de typage des variables.

$$(\textsc{Instance})\frac{instancier\ (acces\ (x, Env)) = ty}{Env \vdash\ x\ :\ ty}$$

avec:

$$instancier\ (\forall\alpha_1, ..., \alpha_n.t) = t[\tau_1/\alpha_1, ..., \tau_n/\alpha_n]$$

Donc, maintenant, l'environnement est une liste d'association *(idf, schéma de type)*.

Le typage de CAML X

Exemple

```
# let id = function x -> x;;
```

```
# id true,id 1;;
```

$$\dfrac{\dfrac{Env \vdash \texttt{id} : \forall\alpha.\alpha \rightarrow \alpha}{Env \vdash \texttt{id} : \texttt{int} \rightarrow \texttt{int}} \quad Env \vdash \texttt{true} : \texttt{bool}}{Env \vdash \texttt{id true} : \texttt{bool}}$$

```
# (function f -> f true,f 1) (function x -> x);;
```


Le typage de CAML XI

Quand et comment généraliser ?

au moment des déclarations

$$\texttt{let}\ idf\ \texttt{=}\ exp1\ \texttt{in}\ exp2$$

On modifie donc un peu la règle de typage des déclarations.

Expression expansive: effectuant des effets de bord pouvant invalider le typage. En CAML, expressions de la forme $e_1\ e_2$.

1. $Env \vdash exp1 : t_1$
2. Si $exp1$ NON expansive, <u>généraliser</u> t_1 en $st_1 = generaliser_{Env}(t_1)$, sinon $st_1 = t_1$
3. Ajout du type obtenu à l'environnement
4. $\lfloor (x : st_1) \triangleleft Env \rfloor \vdash exp2 : ty_2$
5. L'environnement global est inchangé

$$\dfrac{Env \vdash exp_1 : t_1 \quad \lfloor (idf : st_1) \triangleleft Env \rfloor \vdash exp_2 : t_2}{Env \vdash \texttt{let}\ idf\ \texttt{=}\ exp_1\ \texttt{in}\ exp_2 : t_2}$$

Le typage de CAML XII

La généralisation

$$generaliser_{Env}(typ) = \forall \tau_1, ..., \tau_n.typ$$

où les variables τ_i ne sont pas libres dans Env

Déclarations globales:

let id = exp;;

Même principe qu'auparavant mais la liaison ($x : st_1$) est ajoutée à l'environnement global.

Exemple

```
#let id = function x -> x;;

#let f = id id;;

#let g = function x -> (id id) x;;
```


Modularité I

Un exemple

```
#2 + 3;;

#prefix +;;

##open "float";;

#prefix +;;

#1.2 + 1.3 ;;

#2 + 3;;
```

Modularité II

```
# add_int 2 3;;

# #close "float";;

# 2 + 3;;

# 1.2 + 1.3 ;;
```


Modularité III

Un module est défini par:

1. Un corps : définitions (types, fonctions, constantes, etc.) regroupées dans un texte nommé `mod.ml`.
2. Une interface d'exportation nommée `mod.mli` qui définit les déclarations visibles à l'extérieur: contient des déclarations de types et de valeurs (*nom*, type).
3. Pour l'importateur: module = env. de typage (utilisé au cours du typage)
4. si aucune interface `mod.mli` n'est définie, toutes les déclarations sont exportées.

Modularité IV

Structure d'un environnement complet

$$\lfloor D \rfloor \bowtie \lfloor O \rfloor \bowtie \lfloor C \rfloor$$

1. D : liaisons des déclarations du pgm. en cours
2. O : liaisons des modules ouverts
3. C : modules connus du système

Environnement initial *EInit*

1. D : vide
2. O : modules de la librairie "core" `=>` man. de ref.
3. C : modules connus du système

Modularité V

Construire un environnement par importation

1. C : les liaisons déclarées dans les interfaces `mod.mli` connues du système sont accessibles avec la notation `mod__idf`.
2. ouvrir `mod`: placer les liaisons de `mod` au début de O.
3. Si `idf` de `mod` déjà lié dans O `=>` masquage de la liaison antérieure, qui reste accessible avec la notation complète.
4. Aucun masquage des liaisons de D.

Autre choix possible : pas de masquage dans O. Si conflit, notation complète (choix de Ada).

A FAIRE

Vous connaissez un langage de programmation, qui n'est pas Caml-Light. Dans ce langage :

1. Quelle est la syntaxe des déclarations globales? Où doivent-elles être placées?
2. Même question avec les déclarations locales.
3. Si ce langage est typé, comment est faite la vérification/inférence des types.
4. Si ce langage offre des bibliothèques, y a-t-il compilation séparée?
5. Comment sont gérés les noms des bibliothèques?

Traits impératifs I

```
#let counter = ref 0;;

#let add_counter n =
#   counter := !counter + n;;

#let afficher mess =
#   print_string mess;
#   print_newline ();;

#exception Division_par_zero;;

#let division x y =
#   if y = 0 then raise Division_par_zero
#   else x / y;;
```

Traits impératifs II

Référence (`ref` exp)

$$\frac{Env \vdash exp : t}{Env \vdash \texttt{ref}\ exp : t\ \texttt{ref}}$$

Affectation ($exp1$ `:=` $exp2$)

$$\frac{Env \vdash exp1 : t1\ \texttt{ref} \quad Env \vdash \texttt{exp2 : t1}}{Env \vdash exp1 := exp2 : \texttt{unit}}$$

Séquence ($exp1$ `;` $exp2$)

$$\frac{Env \vdash exp1 : t1; \quad Env \vdash exp2 : t2}{Env \vdash exp1; exp2 : \texttt{t2}}$$

Exceptions (`exception` idf `of` typ`;;`)

$$Env \vdash idf : typ \to \texttt{exn}$$

$$Env \vdash \texttt{raise} : \forall\alpha.\texttt{exn} \to \alpha$$

Un algorithme d'unification entre types

Un langage de types:

$$ty \quad ::= \quad ty \to ty \mid ty * ty \mid \texttt{int} \mid var$$

var est un ensemble infini de variables $\alpha_1, \ldots$.

Substitution:

Une substitution est une fonction $s : var \to ty$ étendue aux types telle que:

- $s(t_1 \to t_2) = s(t_1) \to s(t_2)$
- $s(t_1 * t_2) = s(t_1) * s(t_2)$
- $s(\texttt{int}) = \texttt{int}$

Unification:

Deux types t_1 et t_2 sont unifiables, ce que l'on notera $t_1 \equiv t_2$, s'il existe une substitution s telle que:

$$s(t_1) = s(t_2)$$

Un algorithme d'unification entre types

Une substitution s'écrit $[\alpha_1 : t_1, ..., \alpha_n : t_n]$. Connaissant la substitution S_1, l'unification de t_1 et t_2 produit la nouvelle substitution S_2. On le note:

$$S_1 \ \vdash \ t_1 \equiv t_2 \ : \ S_2$$

Avec les règles:

$$S \ \vdash \ t \equiv t \ : \ S$$

$$\frac{S \ \vdash \ t_1 \equiv t'_1 \ : \ S_1 \quad S_1 \ \vdash \ t_2 \equiv t'_2 \ : \ S_2}{S \ \vdash \ t_1 \to t_2 \equiv t'_1 \to t'_2 \ : \ S_2}$$

$$\frac{S \ \vdash \ t_1 \equiv t'_1 \ : \ S_1 \quad S_1 \ \vdash \ t_2 \equiv t'_2 \ : \ S_2}{S \ \vdash \ t_1 * t_2 \equiv t'_1 * t'_2 \ : \ S_2}$$

$$\frac{S \ \vdash \ t_1 \equiv t_2 \ : \ S_2 \quad S(\alpha) = t_1}{S \ \vdash \ \alpha \equiv t_2 \ : \ S_2}$$

$$\frac{\alpha \notin Dom(S) \quad \alpha \notin Vars(S(t))}{S \ \vdash \ \alpha \equiv t \ : \ S[\alpha : S(t)]}$$

Conclusion

- Déclaration du type des identificateurs/synthèse du type
- Vérification/inférence
- Typage fort: une expression a un type unique
- Garanties apportées par le typage fort

Extensions

- surcharge
- sous-typage, objets
- modules

Les types de données

Plan

- Introduction
- types de base
- types produit
- types somme
- types récursifs

Introduction

Des structures:

- contruire des structures complexes à partir de structures de base
- les types donnent une information sur ces structures

Des fonctions:

- elle manipulent (construisent/ accèdent) ces structures de données.
- le filtrage est un moyen de raisonner "par cas" sur la forme de ces structures.

Types de base

Des collections finies d'éléments. Ils correspondent souvent à des valeurs primitives de la machine.

- int, bool,...

Produit Cartésien I

Produit cartésien d'ensembles:
$A \times B = \{(x,y) : x \in A, y \in B\}$

Projections: $A \times B \rightarrow A$, $A \times B \rightarrow B$

$(A \times B) \times C \neq A \times (B \times C)$ mais isomorphisme

Produit cartésien de deux types:

$$\frac{Env \vdash e_1 : t_1 \quad Env \vdash e_2 : t_2}{Env \vdash (e_1, e_2) : t_1 * t_2}$$

```
#fst;;
```

```
#snd ;;
```

Produit cartésien de n types:

`t_1 * t_2 * ..* t_n`

Attention:

`t1 * t2 * t3` $\neq$ `(t1 * t2) * t3`
$\neq$ `t1 * (t2 * t3)`

Projections à définir par l'utilisateur

Produit cartésien II

Paires

Un *constructeur de type* `*` d'arité 2, un *constructeur de valeur* : la virgule `,`

N-uplets

Un *constructeur de type* d'arité n formé de $(n-1)$ `*`, un *constructeur de valeur* formé de $(n-1)$ virgules `,`

Définitions par cas: filtrage

```
#let fst = function (x,y) -> x;;
```

```
#let snd = function (x,y) -> y;;
```

```
#let acceder3 = function (x,y,z,t) -> z;;
```

```
#let double_zero = function
#      (0,0) -> true
#    | (x,y) -> false;;
```

Type Enregistrement I

Un type produit à composantes nommées.

Constructeur de type donné par l'utilisateur
`type ... = {lab_1 : t_1 ; .., lab_n : t_n}`

Constructeur de valeur syntaxe presque identique.

```
#type enfant = {nom : string ; age : int};;
```

```
#let simon = {nom = "Simon" ; age = 9};;
```

Notation pointée

```
#let nom {nom = n;age = a} = n;;
```

```
#let nom {nom = n} = n;;
```

```
#let nom x = x.nom;;
```


Enregistrements III

```
#let anniv = function x ->
#   {nom = x.nom ; age = x.age + 1};;
```

```
#(anniv simon);;
```

```
#simon;;
```

Masquage

```
#type societe = {nom : string; numero : int};;
```

```
#let nom2 x = x.nom;;
```

Règle de typage

$$\frac{Env \vdash \ e \ : \ typ \quad typ = \{lbl_1 : t_1; ...; lbl_n : t_n\}}{Env \vdash \ e.lbl_i \ : \ t_i}$$

Types somme I

Union disjointe d'ensembles:
$t = A \oplus B = \{x.g |\ x \in A\} \ \cup \ \{y.d |\ y \in B\}$

Injection canonique

- $gauche : A \ \ \rightarrow A \oplus B$
- $droit : A \ \ \rightarrow A \oplus B$

On a $x : t$ ssi

$\exists y1 : t1$ tq $x = g(y1)$ XOR $\exists y2 : t2$ tq $x = d(y2)$

En Caml, ces injection sont données par l'utilisateur au moment de la définition du type.

Types somme II

Syntaxe: `type .. = C1 of t1 | C2 of t2`

Convention: `C of unit` ⇒ `C`

```
#type color = Coeur | Pique | Carreau | Trefle ;;
```

```
#type carte = Dam of color
#           | Peti of int*color
#           | Jok ;;
```

`Coeur`, `Dam`, `Peti` constructeurs de valeur.

Prise en compte de la définition On ajoute à l'env. global *Env* les liaisons

$\lfloor$`(C1,t1 -> som)`, `(C2,t2 -> som)` $\triangleleft$ *Env*$\rfloor$

Règle de typage

$$\frac{Env \vdash \ C_i \ : \ t_i \rightarrow som \quad Env \vdash \ e \ : \ t_i}{Env \vdash \ C_i \ e \ : \ som}$$

Types somme III

Filtrage - Somme

```
#let valeurfoo = function
#   | Dam _  -> 20
#   | Peti (10, c) -> 10
#   | Peti (x, y) ->
#       if x = 9 & y = Coeur then 11 else 0
#   | Jok  -> 1 ;;
```

```
#type unites = V of float | H  of float;;
```

```
#let distance_parcourue =
#  function (V x, H y) -> x *. y ;;
```

```
#distance_parcourue (V 25.0, H 4.0);;
```


Types somme IV

Les motifs M sont définis ainsi:

$$
\begin{aligned}
M \quad ::= \quad & C \mid C\ M \mid (M, ..., M) \\
& \mid \texttt{true} \mid \texttt{false} \mid i \\
& \mid x \mid \_
\end{aligned}
$$

où:

- C est un constructeur de type
- i désigne un entier
- x, une variable

Types somme IV

Typage des fonction définies par filtrage

Un cas

$$\frac{\begin{array}{c}\lfloor (x_1 : t_1), ..., (x_n : t_n), Env \rfloor \ \vdash \ M \ : typ_1 \\ \lfloor (x_1 : t_1), ..., (x_n : t_n), Env \rfloor \ \vdash \ exp \ : typ_2\end{array}}{Env \vdash \texttt{function} \ M \ \texttt{->} \ exp : typ_1 \rightarrow typ_2}$$

où les x_i sont les variables du motif M.

Plusieurs cas

$$\frac{\begin{array}{c}Env \vdash \texttt{function} \ M_1 \ \texttt{->} \ exp_1 \ : typ_1 \rightarrow typ_2 \\ ... \\ Env \vdash \texttt{function} \ M_n \ \texttt{->} \ exp_n \ : typ_1 \rightarrow typ_2\end{array}}{\begin{array}{l}Env \vdash \texttt{function} \ |M_1\texttt{->}exp_1 \quad : typ_1 \rightarrow typ_2 \\ \qquad\qquad\qquad\quad ... \\ \qquad\qquad\qquad\quad |M_n\texttt{->}exp_n\end{array}}$$

Types récursifs I

Un type est récursif si sa définition contient au moins une occurrence de son nom.

```
#type exp =   V of string
#           | C of int
#           | P of exp * exp ;;

#let mon_exp = P(V "x", P (V"y", C 3));;

#let rec eval = function
# | V _ -> 0
# | C n  -> n
# | P (e1, e2) -> (eval e1) + (eval e2);;

#let rec deriv = function s -> function
# | V nom -> if s = nom then (C 1) else (C 0)
# | C n  ->  (C 0)
# | P (e1, e2) -> P((deriv s e1),
#                    (deriv s e2));;
```

Types récursifs II

```
# deriv "x" mon_exp;;
```

Attention : Syntaxe correcte $\not\Rightarrow$ sémantique définie

```
# type sans_val = C of sans_val * sans_val ;;
```


Conclusion

- Un programme = des données + des fonctions
- filtrage = raisonnement par cas sur la structure de ces données

Questions

- représentation de ces données
- qu'est ce qu'une valeur?
- comment obtenir la valeur d'une expression?