# Le modèle à enregistrement.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/˜remy/mot/9/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/9/

**Slide 1**

| **Cours** | **Exercices** |
| --- | --- |
| 1. Modélisation du calcul<br>2. Calculs d'objets<br>3. Sémantique<br>4. Codage avec enregistrements<br>5. Typage et sous-typage<br>6. Prototypes<br>7. Encapsulation | 1. Réduction<br>2. Évaluateur du calcul d'objets<br>3. Enregistrements extensibles<br>4. Objets avec des enregistrements |

## Prélude

**Slide 2**

Le but de ce cours est de mieux comprendre ce que sont les objets :
- Quelle est l'essence du mécanisme de la programmation avec objets ?
- Comment peut-on l'étudier formellement ?

Pour cela, nous nous placerons un langage dépouillé de toutes les constructions non essentielles appelé *calcul*.

Nous nous étudierons, succinctement :
- Le lambda-calcul comme modèle de la programmation fonctionnelle.
- Un calcul d'objet.
- Le codage des objets dans un lambda-calcul avec enregistrements.
- Les problèmes de typage.
- L'encapsulation.

## Modélisation du calcul

**Slide 3**

Les calculs sont des simplifications des langages de programmation pour permettre leur étude formelle. Ils ne retiennent que les constructions essentielles (ce qui peut dépendre du contexte d'étude) et font abstraction des détails et des constructions dérivées ou dérivables. Ainsi, les constructions d'un calcul sont en petit nombre et aussi orthogonales (indépendantes) que possible.

En contrepartie, un calcul ne permet en général d'écrire que de petits exemples : pour passer d'un calcul à un langage de programmation, on ajoute du sucre syntaxique et des constructions dérivées permettant d'abréger certaines écritures ou combinaisons fréquentes

## Le lambda calcul

C'est le modèle des langages fonctionnels : il ne retient que les opérations d'abstraction et d'application :

**Slide 4**

| | |
|---|---|
| $e ::= x$ | Variable (ou constante) |
| $\mid c$ | Constante (ou primitive) |
| $\mid \texttt{fun}\ (x)\, e$ | Fonction |
| $\mid e_1\ e_2$ | Application |

On peut bien sûr étendre le lambda-calcul avec quelques types primitifs (entiers, caractères, chaînes, *etc.*), des types construits (tuples, enregistrements), *etc.*

## Réduction

**Slide 5**

On définit la *sémantique opérationnelle* du calcul par réécriture des programmes sur eux-mêmes (c'est l'approche la plus simple).

On identifie un sous-ensemble remarquable des programmes complètement évalués, appelés *valeurs*. Dans le lambda-calcul les valeurs notées par la lettre $v$ sont les fonctions $\texttt{fun}\ (x)\, e$ et les constantes $c$.

Une réduction élémentaire est une réduction en tête, appelé *radical* opérée à un emplacement autorisé appelé *contexte d'évaluation*.

## Réductions en tête (appel par valeur)

**Slide 6**

Une réduction élémentaire en tête est :

– La $\beta$-réduction,

$$(\texttt{fun}\ (x)\ e)\ v \longrightarrow e\{x \leftarrow v\}$$

L'expression $v\{x \leftarrow e\}$ désigne l'expression $v$ dans laquelle toute occurrence de la variable $x$ est remplacée par $v$

– Une $\delta$-réduction est une réduction de la forme

$$c_1\ \overline{c}_2 \longrightarrow c_3$$

pour certains triplets $(c_1, \overline{c}_2, c_3) \in \delta$ Ici $c_1$ doit être une primitive, $\overline{c}_2$ un tuple de constantes et $c_3$ des constantes. Par exemple l'opération successeur est définie en ajoutant l'ensemble des paires $(succ, n, n+1)_{n \in I\!N}$ dans $\delta$.
(On peut facilement généraliser à des opérations binaires.)

## Contextes d'évaluation (appel par valeur)

**Slide 7**

Les emplacements où la réduction est autorisée sont décrits par des contextes d'évaluation, *i.e.* des programmes avec un trou noté $\{\}$, définis par la grammaire :

| | |
|---|---|
| $E ::= \{\}$ | Position où se passe la réduction |
| $\mid E\ e$ | Évaluation à gauche en premier, puis |
| $\mid v\ E$ | Évaluation à droite si |
| | la partie gauche est une valeur |

En appel par valeur, on ne réduit jamais sous une abstraction ; l'argument doit être réduit en une valeur avant de le passer à la fonction.

Ici, on a fixé l'ordre d'évaluation de gauche à droite, ce qui rend la réduction déterministe *a fortiori* (il n'y a qu'une seule chaîne de réécriture possible).

## Évaluation

Une évaluation est une chaîne de réductions élémentaires :

$$e_0 \longrightarrow e_1 \longrightarrow ...e_n$$

Lorsque $e_n$ est irréductible, on ne peut plus étendre la chaîne.

**Slide 8**

## Exemple

$\boxed{(\texttt{fun}\ (x)\, x)\ (\texttt{fun}\ (y)\, y)}\ ((\texttt{fun}\ (z)\, z)\ 3) \longrightarrow$

$(x\{x \leftarrow \texttt{fun}\ (y)\, y\})\ ((\texttt{fun}\ (z)\, z)\ 3) \equiv$

$(\texttt{fun}\ (y)\, y)\ \boxed{((\texttt{fun}\ (z)\, z)\ 3)} \longrightarrow$

$(\texttt{fun}\ (y)\, y)\ (z\{z \leftarrow 3\}) \equiv$

$\boxed{(\texttt{fun}\ (y)\, y)\ 3} \longrightarrow$

$(y\{y \leftarrow 3\}) \equiv 3$

**Exercise 1** *Réduire le terme*
$(\texttt{fun}\ (z)\, \texttt{fun}\ (t)\, \texttt{fun}\ (f)\, (z\ t)\ f)\ (\texttt{fun}\ (x)\, \texttt{fun}\ (y)\, x)\ 1\ 2.$ Answer
*Par quel terme peut-on remplacer* $(\texttt{fun}\ (x)\, \texttt{fun}\ (y)\, x)$ *pour que le résultat final soit* 2 *? Commenter.* Answer □

**Slide 9**

## Exemple (suite)

**Exercise 2** *Donner un exemple de programme erroné (qui ne se réduit pas sans pour autant être une valeur).* Answer

*Donner un exemple de programme qui boucle (qui se réduit indéfiniment).* Answer □

**Slide 10**

## Déterminisme

En général, il existe plusieurs chaînes de réduction possibles.

Une bonne propriété du calcul est son caractère déterministe qui dit que si une chaîne de réduction issue de $e_0$ termine sur une expression irréductible $e_n$, alors toutes les chaînes possibles issues de $e_0$ terminent également (en un nombre fini d'étapes), sur la même expression $e_0$.

**Slide 11**

Ci-dessus, le caractère déterministe est assuré *a fortiori* par la forme des contextes d'évaluation : il n'y a jamais qu'un seule façon d'étendre une chaîne de réduction.

## Terminaison et erreurs

En pratique, on souhaite souvent qu'une réduction se termine sur une valeur $v$ (une valeur est toujours choisie irréductible). Ce qui n'est bien sûr pas garanti.

### Non terminaison

**Slide 12**

Une évaluation peut boucler, *i.e.* elle peut être étendue indéfiniment. (Si le calcul est complet, *i.e.* s'il permet d'écrire tous les algorithmes possibles, alors la terminaison de l'évaluation est indécidable.)

### Erreurs

Une évaluation peut aussi produire une erreur, ce qui est modélisé par une réduction qui termine sur une expression irréductible qui n'est pas une valeur. Par exemple, $(\texttt{fun}\ (x)\,x)\ 2\ (\texttt{fun}\ (y)\,y)$ produit une erreur : $2\,(\texttt{fun}\ (y)\,y)$.

## Pour en savoir plus

Le lambda-calcul est au cœur de la formalisation de la plupart des langages de programmation. Ses propriétés formelles ont été étudiés très en détail.

Il peut être muni de différentes stratégies d'évaluation (appel par nom, évaluation paresseuses) modélisant l'évaluation dans différents langages, et d'une évaluation dite forte permettant de réduire sous les abstractions modélisant par exemple les optimisations à la compilation.

**Slide 13**

Pour en savoir plus, voir
- Les cours de compilation en Majeure 2
- Le D.E.A. Programmation : Sémantique, Preuves et Langages

## Calculs d'objets

Slide 14

Le lambda-calcul ne rend pas bien compte de la notion d'objet, en particulier des appels récursifs et de la liaison tardive.

C'est pour pouvoir étudier directement les objets que des calculs dédiés ont été introduits. Seules les constructions essentielles des langages à objets sont retenues.

Par exemple, les variables d'instance ne seront pas modélisées (elles peuvent s'écrire comme des méthodes qui n'utilisent pas `self`). Un calcul simplifié va aussi ignorer les champs mutables, du moins dans un premier temps. Ceux-ci peuvent ensuite être facilement réintroduits dans un calcul étendu ou dans un vrai langage.

L'abstraction et l'application ne sont pas des constructions primitives, parce qu'elles seront codables par des objets.

## Un calcul d'objets (non typé)

Slide 15

Ce calcul est dû à Abadi-Cardelli.

Un objet est un petit enregistrement de méthodes pouvant s'appeler récursivement.

| | |
|---|---|
| $e ::= x$ | Variable |
| $\mid \langle \ell_i = \varsigma\,(x_i)\,e_i \rangle_{i=1}^{n}$ | Objet |
| $\mid e.\ell$ | Envoi de message |
| $\mid e.\ell \Leftarrow \varsigma\,(x)\,e'$ | Modification/ajout d'une méthode |

Une méthode $\varsigma\,(x)\,e$ lie la variable $x$ à *self*. Ici, chaque méthode peut utiliser un nom différent pour *self*. La construction $p.\ell \Leftarrow \varsigma\,(x)\,e$ remplace (ou ajoute) la méthode $\ell$ dans $p$ avec la définition $\varsigma\,(x)\,e$.

N.B. : on considère les objets modulo commutation des champs.

## Exemple (dans le calcul avec fonctions)

Un point (approche fonctionnel) :

$$\begin{aligned} p \triangleq \langle \quad & x = 1, \\ & getx = \varsigma\,(z)\,.z.x, \\ & setx = \varsigma\,(z)\,\texttt{fun}\ (y)\,(z.x \Leftarrow y) \quad \rangle \end{aligned}$$

**Slide 16**

Un point qui bouge (liaison tardive) :

$$\begin{aligned} q \triangleq ( \quad & p. \\ & bouge \Leftarrow \varsigma\,(z)\,(z.setx(z.getx + 1)) \quad ) \end{aligned}$$

Le point qui bouge s'évalue (voir les règles ci-dessous) en

$$\begin{aligned} q \longrightarrow \langle \quad & x = 1, \\ & getx = \varsigma\,(z)\,.z.x, \\ & setx = \varsigma\,(z)\,\texttt{fun}\ (y)\,(z.x \Leftarrow y) \\ & bouge = \varsigma\,(z)\,(z.setx(z.getx + 1)) \quad \rangle \end{aligned}$$

## Exemple (suite)

L'expression $p.bouge$ s'évalue en le nouvel objet :

$$\begin{aligned} q \longrightarrow \langle \quad & x = 2, \\ & getx = \varsigma\,(z)\,.z.x, \\ & setx = \varsigma\,(z)\,\texttt{fun}\ (y)\,(z.x \Leftarrow y) \\ & bouge = \varsigma\,(z)\,(z.setx(z.getx + 1)) \quad \rangle \end{aligned}$$

**Slide 17**

et $(p.bouge).bouge.getx$ s'évalue en 3.

## Sémantique opérationnelle

**Les valeurs** sont les objets.

**Réduction élémentaire**

On note $m$ une collection de méthodes $(\ell_i = \varsigma\,(x_i)\,e_i)_{i \in I}$.

Slide 18

$$\begin{aligned}
\langle\, \ell = \varsigma\,(x)\,e; m \,\rangle.\ell &\longrightarrow e\{x \leftarrow \langle\, \ell = \varsigma\,(x)\,e; m \,\rangle\} \\
\langle\, \ell = \varsigma\,(x)\,e; m \,\rangle.\ell \Leftarrow \varsigma\,(x')\,e' &\longrightarrow \langle\, \ell = \varsigma\,(x')\,e'; m \,\rangle \\
\langle\, m \,\rangle.\ell \Leftarrow \varsigma\,(x')\,e' &\longrightarrow \langle\, \ell = \varsigma\,(x')\,e'; m \,\rangle \qquad l \notin dom\,(m)
\end{aligned}$$

**Contextes d'évaluation** (évaluation en tête)

$$E ::= \{\} \mid E.\ell \Leftarrow \varsigma\,(x)\,e \mid E.\ell$$

## Exemple d'évaluation

On omet $\varsigma\,(x)$ dans $\varsigma\,(x)\,e$ quand $x$ n'apparaît pas dans $e$.

Slide 19

$$\begin{aligned}
&\boxed{(\langle\, f = \varsigma\,(x)\,x.a.\mathtt{geth} \,\rangle.a \Leftarrow \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle)}.f \longrightarrow \\
&\langle\, f = \varsigma\,(x)\,x.a.\mathtt{geth}; a = \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle \,\rangle.f \equiv \\
&\boxed{\langle\, f = \varsigma\,(x)\,x.a.\mathtt{geth}; a = \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle \,\rangle.f} \longrightarrow \\
&(x.a.\mathtt{geth})\{x \leftarrow \langle\, f = \varsigma\,(x)\,x.a.\mathtt{geth}; a = \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle \,\rangle\} \equiv \\
&\boxed{\langle\, f = \varsigma\,(x)\,x.a.\mathtt{geth}; a = \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle \,\rangle.a}.\mathtt{geth} \longrightarrow \\
&\boxed{\langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle.\mathtt{geth}} \longrightarrow \\
&(x.h)\{x \leftarrow \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle\} \equiv \\
&\boxed{\langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h \,\rangle.h} \longrightarrow \\
&3
\end{aligned}$$

**Exercise 3 (Évaluateur du calcul d'objets)** *Le but de cet exercice* *est d'écrire un évaluateur pour le calcul d'objets.*

*On représente les expressions du langage par le type Ocaml suivant :*

**Slide 20**

```
type expr =
  (* Variables *)
  | Var of variable | Int of int
  (* Objets *)
  | Obj of (label * methode) list
  (* Envoi de message *)
  | Mes of expr * label
  (* Modification/Ajout de méthode *)
  | Mod of expr * label * methode
and methode = variable * expr
and variable = string
and label = string
;;
```

*Par exemple,* $\ell = \varsigma\,(x)\,x.m$ *est représenté par le terme :*
`Obj [ "l", ("x", Mes (Var "x", "m")) ]`. *Définir, en Ocaml, le terme représentant le programme :*

$$(\langle\, f = \varsigma\,(x)\,x.a.\mathtt{geth}\,\rangle.a \Leftarrow \varsigma\,(\_)\,\langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h\,\rangle).f$$

Answer

**Slide 21**

*Définir un évaluateur pour le calcul.*

*(C'est-à-dire définir une fonction récursive `eval` qui prend en arguments un environnement d'évaluation `env` une expression `expr` et retourne le résultat de l'évaluation de l'expression `expr` dans l'environnement `env`. L'environnement d'évaluation est une liste d'association entre des variables et leur valeur. L'environnement représente la substitution qu'il faudrait appliquer au terme à évaluer.)* Answer □

## Héritage (simple) dans le calcul d'objet

Dans le calcul d'objet, l'héritage (simple) est modélisé par la (re)définition de méthodes.

Par exemple on peut fabriquer un bipoint $q$ à partir d'un point $p$ :

$$\begin{array}{lcl} p & \triangleq & \langle\, x = 3; \texttt{getx} = \varsigma\,(z)\, z.x \,\rangle \\ q & \triangleq & (p.y \Leftarrow 5).\texttt{gety} \Leftarrow \varsigma\,(z)\, z.y \end{array}$$

**Slide 22**

Ce calcul ne modélise pas directement l'héritage multiple.

## Codage des fonctions (non typé)

Une fonction est un objet avec une méthode qui contient le corps de la fonction :

$$[\![\texttt{fun}\ (x)\, e]\!] = \langle\, \texttt{fun} = \varsigma\,(x)\,[\![e]\!]\,\rangle$$

La valeur de l'argument sera mémorisé à coté du code de la fonction dans un champ `arg`. Aussi, l'accès à une variable devient :

**Slide 23**

$$[\![x]\!] = x.\texttt{arg}$$

L'application place l'argument à coté de la fonction (fabrication d'une fermeture) et lance l'exécution.

$$[\![e_1\ e_2]\!] = ([\![e_1]\!].\texttt{arg} \Leftarrow \varsigma\,(x)\,[\![e_2]\!]).\texttt{fun}$$

NB : ce codage est en appel par nom (l'argument n'est pas évalué avant d'être remplacé dans le corps de la fonction).

## Ajout de champs valeur

On ajoute une construction $e.\ell \Leftarrow e'$ (ici $e'$ ne dépend pas de *self*), avec pour règle de réduction, et contextes d'évaluations :

$$(v.\ell \Leftarrow v) \longrightarrow (v.\ell \Leftarrow \varsigma\,(x)\,v) \qquad E ::= \ldots \mid E.\ell \Leftarrow e \mid v.\ell \Leftarrow E$$

**Slide 24**

À la différence des méthodes, les champs sont évalués avant d'être ajoutés à l'objet.

L'application en appel par valeur peut être codée par :

$$[\![e_1\ e_2]\!] = ([\![e_1]\!].\mathtt{arg} \Leftarrow [\![e_2]\!]).\mathtt{fun}$$

*NB : En Ocaml, les champs ne sont pas abstraits par rapport à* self *justement parce qu'ils sont évalués avant d'être ajoutés à l'objet ! Pour voir self dans un champ, il faut abstraire par rapport à self, mais cela a aussi pour effet de geler l'évaluation.*

## Objets $\approx$ fermetures

**Slide 25**

**Codage de l'application** $[\![(\mathtt{fun}\ (x)\,e_1)\ e_2]\!]$

Juste avant d'être lancé, l'application devient un objet

$$\langle\, \mathtt{fun} = \varsigma\,(x)\,.[\![e_1]\!], \mathtt{arg} = [\![e_2]\!]\,\rangle$$

C'est la fermeture du corps de la fonction avec la valeur de l'argument (voir le cours langage et programmation ou le cours de compilation).

**Fonctions mutuellement récursives** (à plusieurs arguments)

Elles sont compilées vers un objet unique :
- les variables libres (définies en dehors du corps de la fonction) sont placés dans des variables d'instances.
- les fonctions sont des méthodes.

Le corps de chaque fonction a accès aux autres fonctions et aux variables libres.

## Codage dans le lambda-calcul

On montre ici le codage réciproque du calcul d'objet dans le lambda-calcul. L'intérêt est de vérifier le slogan :

*Les objets sont des enregistrements de fonctions !*

Toutefois, il faut au préalable étendre le lambda-calcul avec des enregistrements extensibles.

**Slide 26**

## Calcul avec enregistrements extensibles

Pour modéliser les objets, on ajoute des enregistrements extensibles. On se donne un ensemble dénombrable d'étiquettes, et trois nouvelles constructions :

| | |
|---|---|
| $e ::= \ldots$ | Comme dans le lambda-calcul |
| $\mid \{\, \ell_i = e_i \,\}$ | Enregistrement |
| $\mid \{\, x \ \mathtt{with}\ \ell = e \,\}$ | Ajout ou modification de champ |
| $\mid x.\ell$ | Accès |

**Slide 27**

Les enregistrements sont des fonctions partielles des étiquettes vers les valeurs (comme des tables d'association où les clés seraient des étiquettes).

## Sémantique des enregistrements

Slide 28

On ajoute $\{ \ell_i = v_i \}$ parmi les valeurs. On considère les valeurs-enregistrements modulo commutation des champs.

### Contextes d'évaluation

| | |
|---|---|
| $E ::= \dots$ | Comme avant |
| $\mid \{ \ell_i = v_i; \ell = E; \ell_j = e_i \}$ | On évalue de gauche à droite |
| $\mid \{ E \texttt{ with } \ell = e \}$ | On évalue à gauche, |
| $\mid \{ v \texttt{ with } \ell = E \}$ | Puis à droite |
| $\mid E.\ell$ | |

**Radical** (soit $m$ de la forme $(l_i = v_i)_{i \in I}$)

$$\{ \ell = v; m \}.\ell \longrightarrow v$$

$$\{ \{ \ell = v; m \} \texttt{ with } \ell = v' \} \longrightarrow \{ \ell = v'; \ell_i = v_i \}$$

$$\{ \{ m \} \texttt{ with } \ell = v \} \longrightarrow \{ \ell = v; m \} \qquad (\text{si } \ell \neq \ell_i)$$

Slide 29

**Exercise 4 (Enregistrements extensibles finis)** *Dans cet exercice, on veut faire découvrir le mécanisme de typage des enregistrements extensibles (sous-jacent au mécanisme de typage des objets). On se limite à un nombre fini de champs.*

*Pour simplifier, on se limite au cas de deux champs* `a` *et* `b`*. On définit le type enregistrements*

```
type ('a, 'b) ab = { a : 'a; b : 'b };;
```

*En utilisant le type option, donner des une valeur représentant* `a1` *correspondant à l'enregistrement* `{ a = 1 }` *définit sur une seul champ, puis une valeur* `a2` *représentant l'enregistrement* `{ a = 1; b = true }` *.* Answer

*Écire la fonction* `get_a` *qui accède au vrai contenu du champ* `a` *en supposant qu'il soit présent. Quel est le problème rencontré (par rapport au typage) ?* Answer

*On remplace le type option par les deux types indépendants suivant :*

```
type 'a pre = Pre of 'a;;
type abs = Abs;;
```

*Redéfinir* `a1` *et* `a2` *en utilisant ces types plutôt que le type option.* Answer

*Écrire la fonction* `get_a` *qui accède au champ* `a` *seulement s'il est présent, et la fonction* `with_a` *qui ajoute à un enregistrement le champ* `a` *(comme la construction* $\{\, \{\, m \,\}\ \texttt{with}\ \ell = v \,\}$*) ? Définir les valeurs correpondantes pour le champ* `b` *ainsi que la valeur correspondant à l'enregistrement vide.* Answer

**Slide 30**

*Reconstuire la valeur* `{ a = 1; b = true }` *à partir des primitives précédentes :* Answer □

## Codage des objets (non typé)

**Slide 31**

On peut coder les objets dans le lambda-calcul avec enregistrements extensibles comme des enregistrements de fonctions.

$$[\![\langle\, \ell_i = \varsigma\,(x_i)\, e_i \,\rangle]\!] = \{\, \ell_i = \texttt{fun}\ (x_i)\ [\![e_i]\!] \,\}$$

La (re)définition de méthode est une redéfinition de champ :

$$[\![e.\ell = \varsigma\,(x)\, e']\!] = \{\, [\![e]\!]\ \texttt{with}\ \ell = \texttt{fun}\ (x)\ [\![e']\!] \,\}$$

L'envoi de message est une auto-application :

$$[\![e.\ell]\!] = [\![e]\!].\ell\ \ [\![e]\!]$$

*NB : Le codage de l'héritage dans le calcul d'objet fourni donc également un codage de l'héritage simple dans les enregistrements extensibles. Pour l'héritage multiple, il faut une opération de concaténation des enregistrements, primitive ou simulée.*

**Exercise 5 (Codage des objets avec des enregistrements)**

*Comme on ne possède pas des enregistrements extensibles, on se contente des enregistrements simples.*

*Définir un type enregistrement permettant de coder un objet point comme ci-dessus (avec champ `x` et deux méthodes `getx` et `setx`). Puis donner une fonction qui prend un entier `x0` et retourne le point d'abcisse `x0` :* Answer

*Définir une fonction `getx` qui prend un point et lui envoie le message `getx` ainsi qu'une fonction `setx` définie de façon similaire.* Answer

**Slide 32**

*Quel serait le type `bpoint` d'un point avec une méthode `bouge` ? Pourquoi ne peut-on pas réutiliser le code de la fonction `point` ci-dessus pour créer un point qui bouge.* Answer

*On pourra essayer de corriger ces deux problèmes en utilisant des enregistrement extensibles (par exemple, en se limitant aux 4 champs `x`, `getx`, `setx` et `bouge`).* □

## Problèmes de typage

Jusqu'à présent, nous avons complètement ignoré le typage.

Le but d'un système de typage est d'éliminer tout risque d'évaluation bloquée. Le prix est une restriction de l'expressivité du langage.

Il existe différents systèmes de typage pour les objets ayant des une expressivité différente.

**Slide 33**

Certaines constructions, indépendemment correctement typées peuvent entré en conflit et être mal typées lorsqu'elles cohabitent.

### Sous-typage en largeur et méthodes binaires

Conflit ! Le sous-typage en largeur (oubli de champs) en présence de méthode binaire n'est pas correct en général (voir *Les points difficiles de la programmation avec objets.*

## Problèmes de (sous-)typage

### Sous-typage en profondeur et redéfinition de méthodes

Conflit ! Une méthode $m_1$ peut utiliser une méthode $m_2$ avec un type $t$. Par sous-typage en profondeur, on peut affaiblir le type $t$ en un type $t'$. La redéfinition de la méthode $m_2$ avec le type $t'$ serait incorrecte car elle briserait l'hypothèse faite par $m_1$.

**Slide 34**

### Sous-typage en largeur et ajout de méthodes

Conflit ! Une méthode $m_1$ peut utiliser une méthode $m_2$ avec un type $t$. Par sous-typage en largeur, on peut oublier que $m_2$ est définie, et la redéfinir avec une méthode $m_3$ d'un autre type $t'$, ce qui briserait l'hypothèse faite par $m_1$.

## Problèmes de typage (solutions)

### Interdire certaines combinaisons

On peut interdire globalement et simultanément
- Le sous-typage en profondeur ou la redéfinition de méthode,
- Le sous-typage en largeur ou l'ajout de méthode.

### Enrichir l'information de type

**Slide 35**

On peut garder des informations plus précises sur les types et distinguer les méthodes selon qu'elles sont (1) implémentées, (2) utilisées dans la vue interne, ou (3) utilisées dans la vue externe. De même on peut se souvenir si les contraintes sont exactes ou si le sous-typage a été utilisé, etc.

Cela donne des systèmes de types plus précis, mais aussi en général trop complexes...

## Problèmes de typage (prototypes)

### Distinguer objets et prototypes

Slide 36

Pour pallier au problème ci-dessus, on peut distinguer :
- Les prototypes : ce sont des objets auxquels on peut ajouter des méthodes, mais pas envoyer de message, ni les sous-typer.
- Les objets : on ne peut plus ajouter de méthodes, mais on peut envoyer des messages et les sous-typer.

On passe d'un prototype à un objet par sous-typage (opération irréversible).

Solution en peigne !

Malgré tout, cela ne résout pas tous les problèmes.

## Les vues

### Utilisation de vues

Slide 37

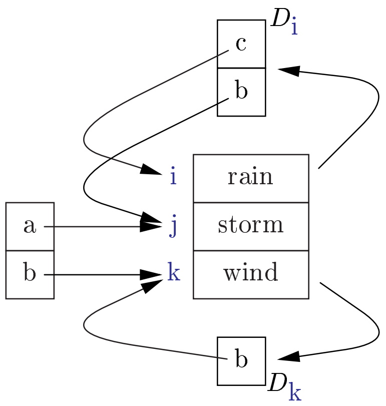
Elles permettent de renommer les noms de méthodes et ainsi de les dissocier de leur implémentation. Un même nom peut avoir plusieurs implémentations. On peut oublier un nom, et le redéfinir, mais avec une nouvelle implémentation (sans écraser l'implémentation précédente).

*NB : Les langages avec vues sont encore compliqués ou limités, et la conception d'un langage avec vues simple et expressif reste du domaine de la recherche...*

Les solutions consistent à ajouter une indirection supplémentaire ; on distingue donc les noms de méthodes et leur emplacement d'une part, l'allocation d'un emplacement et sa définition d'autre part.

## Méthodes relogeables

**Slide 38**




Un objet $p \triangleq d_D$ où $D$ est le dictionnaire externe, $D_\mathrm{i}$ et $D_\mathrm{k}$ sont des dictionnaires internes.

$$d \triangleq \begin{pmatrix} \mathrm{i} = [\mathrm{c} \mapsto \mathrm{i}, \mathrm{b} \mapsto \mathrm{j}], & \text{rain} \\ \mathrm{j} = \ldots, & \text{storm} \\ \mathrm{k} = [\mathrm{b} \mapsto \mathrm{k}], & \text{wind} \end{pmatrix}$$

$$D \triangleq [\mathrm{a} \mapsto \mathrm{j}, \mathrm{b} \mapsto \mathrm{k}]$$

## Problème de typage (codage)

Le codage des objets avec des types enregistrement ne préserve pas le typage : la tranformation d'une méthode en une fonction abstraite par rapport à self en fait une méthode binaire, ce qui interdit la possibilité de sous-typage.

Il existe des codages plus compliqués qui préservent le typage.

**Slide 39**

## Encapsulation

**Slide 40**

L'encapsulation des données est la possibilité de cacher l'état interne des objets, *i.e.* la représentation des données et de ne montrer que leur interface vers le monde extérieur.

C'est une composante importante des objets que nous n'avons pas encore détaillée. Il y a principalement trois approches :
– Cacher les variables d'instance.
– Cacher leur type, en utilisant des types existentiels.
– Cacher leur type, par sous-typage en largeur.
Une dernière solution est de ne pas avoir d'encapsulation... C'est le cas par exemple de l'approche avec surcharge. L'encapsulation peut également être laissée au langage de modules.

## Encapsulation (par abstraction de valeur)

**Slide 41**

C'est l'approche la plus facile : les variables d'instances ne sont jamais accessibles de l'extérieurs, de façon analogue aux liaisons locales :

```
let résultat =
  let x = 1 and y = 2 in
  x + y;;
```

Cette approche impose que les variables d'instances soient une construction primitive. Elle est compliquée par rapport aux liaisons locale par la possibilité d'héritage.

C'est la solution retenue en Ocaml.

En Java, l'approche correspond à cacher les valeurs, mais elle est plus complexe : elle utilise la notion de package, qui se situe plus au niveau des modules que des classes.

## Encapsulation (par abstraction de type)

Un objet est une paire $(r, m)$ composée des variables d'instance $I$ et des méthodes $M$.

Dans la classe *self* a un type de la forme $R \times M$.

L'état interne est caché à l'extérieur, *i.e.* un objet de cette classe a un type existentiel de la forme $\exists(R)(R \times M)$.

**Slide 42**

Toutefois, les méthodes de $M$ dépendent aussi du type de $R$ et simultanément du type externe de l'objet ; la vérité est donc un peu plus compliquée... `rec` $\alpha.\exists(R)(R \times M(R, \alpha))$

Cette méthode est utilisée pour le codage des objets avec des enregistrements.

## Encapsulation (par sous-typage)

On utilise le sous-typage en largeur. C'est l'approche retenue dans le calcul d'objet.

Elle ne s'applique qu'en l'absence de méthodes binaires, ou bien il faut utiliser un système de typage plus riche pour garantir que la variable d'instance n'a pas été utilisée de façon externe à l'intérieur de la class (*i.e.* en accédant à un objet qui n'est pas self mais du même type que self).

**Slide 43**

# 1 Solutions des exercices

## Exercise 1, page 8

$$\begin{array}{l}
\boxed{(\texttt{fun }(z)\,\texttt{fun }(t)\,\texttt{fun }(f)\,(z\ t)\ f)\ (\texttt{fun }(x)\,\texttt{fun }(y)\,x)}\ 1\ 2 \longrightarrow \\
((\texttt{fun }(t)\,\texttt{fun }(f)\,(z\ t)\ f)\{z \leftarrow \texttt{fun }(x)\,\texttt{fun }(y)\,x\})\ 1\ 2 \equiv \\
\boxed{(\texttt{fun }(t)\,\texttt{fun }(f)\,(\texttt{fun }(x)\,\texttt{fun }(y)\,x)\ t\ f)\ 1}\ 2 \longrightarrow \\
((\texttt{fun }(f)\,((\texttt{fun }(x)\,\texttt{fun }(y)\,x)\ t)\ f)\{t \leftarrow 1\})\ 2 \equiv \\
\boxed{(\texttt{fun }(f)\,(\texttt{fun }(x)\,\texttt{fun }(y)\,x)\ 1\ f)\ 2} \longrightarrow \\
(((\texttt{fun }(x)\,\texttt{fun }(y)\,x)\ 1)\ f)\{f \leftarrow 2\} \equiv \\
(\boxed{(\texttt{fun }(x)\,\texttt{fun }(y)\,x)\ 1}\ 2) \longrightarrow \\
((\texttt{fun }(y)\,x)\{x \leftarrow 1\})\ 2) \equiv \\
\boxed{(\texttt{fun }(y)\,1)\ 2} \longrightarrow \\
1\{y \leftarrow 2\} \equiv 1
\end{array}$$

On peut vérifier le résultat de ce calcul en évaluant cette expression écrite dans la syntaxe Ocaml :

```
(fun z t f -> (z t) f) (fun x y -> x) 1 2;;
```

*− : int = 1*

### Exercise 1 (continued)

Le terme $(\texttt{fun }(x)\,\texttt{fun }(y)\,y$ convient. En effet :

$$\begin{array}{l}
\boxed{(\texttt{fun }(z)\,\texttt{fun }(t)\,\texttt{fun }(f)\,(z\ t)\ f)\ (\texttt{fun }(x)\,\texttt{fun }(y)\,y)}\ 1\ 2 \longrightarrow \\
\ldots \\
(\boxed{(\texttt{fun }(x)\,\texttt{fun }(y)\,y)\ 1}\ 2) \longrightarrow \\
((\texttt{fun }(y)\,y)\{x \leftarrow 1\})\ 2) \equiv \\
\boxed{(\texttt{fun }(y)\,y)\ 2} \longrightarrow \\
y\{y \leftarrow 2\} \equiv 2
\end{array}$$

On vérifie :

```
(fun z t f -> (z t) f) (fun x y -> y) 1 2;;
```

*− : int = 2*

Le terme $(\texttt{fun }(z)\,\texttt{fun }(t)\,\texttt{fun }(f)\,(z\ t)\ f)$ se comporte comme `if _ then _ else _` et les termes $(\texttt{fun }(x)\,\texttt{fun }(y)\,x)$ et $(\texttt{fun }(x)\,\texttt{fun }(y)\,y)$ comme `true` et `false`, respectivement.

## Exercise 2, page 10

Le programme 1 2.

### Exercise 2 (continued)

Le programme $(\texttt{fun }(x)\,x\ x)\ (\texttt{fun }(x)\,x\ x)$. En effet il se réduit en tête en une étape sur lui-même.

$$\begin{array}{l}
\boxed{(\texttt{fun }(x)\,x\ x)\ (\texttt{fun }(x)\,x\ x)} \longrightarrow \\
((x\ x)\{x \leftarrow \texttt{fun }(x)\,x\ x\}) \equiv \\
\boxed{(\texttt{fun }(x)\,x\ x)\ (\texttt{fun }(x)\,x\ x)} \longrightarrow \\
\ldots
\end{array}$$

## Exercise 3, page 21

```
let e =
  Mes
   (Mod
     ((Obj [ "f", ("x", Mes (Mes (Var "x", "a"), "geth")) ]), "a",
      ("_",
       (Obj ["h", ("_", Int 3); "geth", ("x", Mes (Var "x", "h"))]))),
    "f");;
```

### Exercise 3 (continued)

```
exception Erreur of string;;
let rec eval env =
  function
   | Var x ->
      begin try List.assoc x env with
      | Not_found -> raise (Erreur ("Variable " ^x^ " libre"))
      end
   | Mes (e, l) ->
      begin match eval env e with
      | Obj p as v ->
         let (x, a) =
           try List.assoc l p
           with Not_found -> raise (Erreur ("Message " ^l^ " not found "))
         in eval ((x, v)::env) a
      | Int 1 -> raise (Erreur "Message à un entier")
      | _ -> assert false
      end
   | Mod (e, l, b) ->
      begin match eval env e with
      | Obj p as v ->
         Obj ((l, b) :: (List.filter (function m, _ -> l <> m) p))
      | Int 1 -> raise (Erreur "Modification d'un entier")
      | _ -> assert false
      end
   | Int n -> Int n
   | Obj p -> Obj p
       ;;
```

On peut vérifier :

```
eval [] e;;
```

*− : term = Int 3*

## Exercise 4, page 29

```
let a1 = { a = Some 1; b = None };;
let a2 = { a = Some 1; b = Some true };;
```

### Exercise 4 (continued)

```
let get_a x =
  match x with Some v -> v | None -> failwith \lst"get_a"
```

Le type `'a option` du n'indique pas si le champ est réellement présent ou absent. Il faut donc tester, et lever une erreur si le champ n'est pas présent.

### Exercise 4 (continued)

```
let a1 = { a = Pre 1; b = Abs };;
let a2 = { a = Pre 1; b = Pre true };;
```

**Exercise 4 (continued)**

```
let get_a r = let (Pre x) = r.a in x;;
let get_b r = let (Pre x) = r.b in x;;
let with_a r x = { a = Pre x; b = r.b };;
let with_b r x = { a = r.a; b = Pre x };;
let vide = { a = Abs; b = Abs };;
```

**Exercise 4 (continued)**

```
let a2 = with_b (with_a vide 1) true;;
```

## Exercise 5, page 32

```
type point =
    {x : int; getx : point -> int; setx : point -> int -> point };;
let point x0 =
  { x = x0;
    getx = (fun self -> self.x);
    setx = (fun self y -> {self with x = y});
  } ;;
```

**Exercise 5 (continued)**

```
let getx p = p.getx p;;
let setx p = p.setx p;;
let p = point 1 in
getx (setx p 2);;
```

*– : int = 2*

**Exercise 5 (continued)**

```
type bpoint
    { x : int; getx : bpoint -> int; setx : bpoint -> int -> bpoint;
      bouge : bpoint -> bpoint };;
```

On pourrait être tenté d'écrire :

```
let bpoint x0 =
  let super = point x0 in
  { x = super.x0; getx = super.getx; setx = super.setx;
    bouge = (fun self -> self.setx self (self.getx self + 1));}
```

Cependant, il y a au moins deux problèmes :

- Le type de self dans `super` est `point` et non `bpoint`.
- La méthode `setx` retourne un `point` et non un `bpoint`.