Modularité, Objets et Types

Cours de Majeure 1
de l'École Polytechnique

Didier Rémy

Octobre 2001

Résumé

Pour comprendre les principes et maîtriser la pratique de la programmation avec objets, les constructions de haut niveau : *objet, classe, héritage, liaison tardive* sont décomposées en des mécanismes plus élémentaires : *enregistrement, fermeture, abstraction, récursion*, étudiés au travers de différentes notions de typage : *statique, dynamique, surchargé, sous-typage, polymorphisme* et sous l'angle de la programmation modulaire : *héritage simple et multiple, modules, mixins.*

Le cours s'appuie principalement sur les langages *Java* et *Ocaml*.

Note

Ce recueil est composé des transparents du cours de compilation de l'année 2000, disponibles électroniquement à l'adresse

http://cristal.inria.fr/˜remy/poly/mot/

Table des matières

Modularité, Objets et Types.

Calendrier des cours

Tous les mercredi

Mois	Octobre					Novembre				Décembre	
EA	3	10	17	24	31	7	14	21	28	5	17–19
Projet							14	21	28	5	

Slide 1

- Cours EA de 13h45 à 15h15 en PC 38
- Travaux dirigés EA de 15h30 à 17h30 en salle info 32.
- Projets EA de 10h15 à 12h15 en salle info 32 ?

Programmation traditionnelle

Slide 2

Les programmes sont écrits de façon linéaire.

On peut seulement ajouter de nouvelles définitions à la fin du programme.

Réutilisation du code par “couper-coller”.

Pas de partage de code
- duplication du code, et des erreurs !
- difficulté de maintenance.

La modularité

Slide 3

Un problème essentiel qui doit être pris au sérieux et considérer dans son ensemble.

Un problème difficile tant en théorie et en pratique.

Des solutions divergentes

Malgré de fortes ressemblances, des motivations presqu'identiques et l'utilisation de presque les mêmes ingrédients, les approches divergent rapidement. Classes ou modules ? Et dans l'approche à classes, les méthodes sont elles portées par les objets ou en dehors comme des fonctions surchargées ?

Cours et projet
- Le cours présente et explique les différents mécanismes disponibles.
- Le projet permet leur mise en œuvre en vraie grandeur.

Quelques mots clés

Abstraction Compositionalité

Extension

Slide 4 Réutilisation Partage

Composants Interface Cacher

Sûreté Typage

Une courte introduction à la modularité

Cette introduction est volontairement générale et peu technique. Elle présente intuitivement, donc très informellement, les différents mécanismes.

– La modularité en général et dans les langages de programmation.
– Sécurité, abstraction, typage.
– La modularité du pauvre
– Les systèmes de modules
– Langages à objets et à classes

Slide 5

La modularité, un concept général

Découpage de l'ensemble en composants indépendants

Rendre les gros projets réalisables

Donner de la structure à l'ensemble

Rendre les gros projets compréhensibles

Spécifier les liens entre les composantes

Slide 6

Rendre les gros projets maintenables

Identifier des sous-composantes indépendantes

Rendre les projets réutilisables

Enfin, en forçant l'abstraction,

La modularité augmente la sûreté.

En informatique remplacer projet par programme.

La modularité dans l'industrie

Les Légos, le Mécano : on construit des objets à partir de briques élémentaires universelles. On peut réutiliser des objets pour produire des assemblages et des mécanismes encore plus complexes.

Dans la mécanique : visserie, roulements à billes ont une interface (nomenclature) très précise pour être interchangeables, réutilisables. Fabrication si besoin d'outils spécialisés ($\approx$ librairies)

Slide 7

Circuits électriques : on forme des opérateurs logiques à partir de transistors, puis des modules (aditioneur, multiplicateur) que l'on assemble ensuite pour formée des unités arithmétiques, etc. Cette composition hiérarchique des circuits électroniques est essentielle.

La modularité en mathématiques

Généraliser pour réutiliser

Résolution d'une équation du second degré : factorisation "intuitive" ou traitement du cas général et application au(x) cas particulier(s).

Un théorème ou une théorie sont souvent étudiés dans le cadre le plus général et on déduit des corollaires pour des cas particuliers.

Slide 8

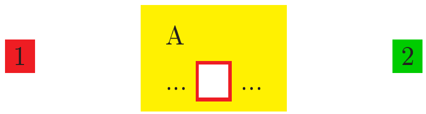
Généraliser pour simplifier

- Disparition des détails (bruit), rendant l'essentiel apparent.
- Meilleure appréhension de l'ensemble (vérification, évolution).

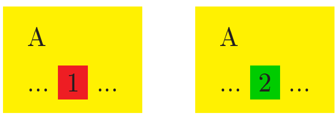
Mais il ne faut pas généraliser... pour la beauté du résultat !

L'abstraction en vaut-elle la peine ? (nombre d'applications, simplification conceptuelle).

Abstraction

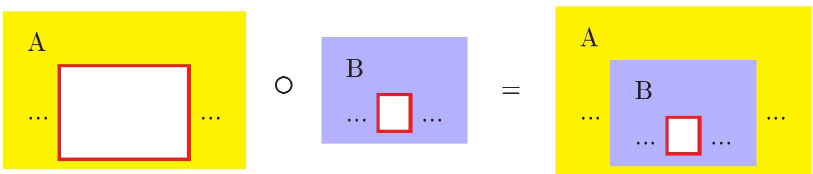


Plusieurs instances partagent le même corps (preuve, code) :



Slide 9

Les abstractions peuvent être composées :



Par exemple à partir d'une algèbre sur un anneau et de l'anneau des polynômes, on peut former l'algèbre des polynômes.

Abstraction *v.s.* Extension

L'abstraction se fait *a priori* (il faut deviner à l'avance les besoins de partage)

Sait-on toujours ce qu'il faut abstraire ?

On ne peut pas tout abstraire...

Slide 10

L'extension se fait *a posteriori*

Par exemple, pour spécialiser une preuve, il faut peut-être ajouter des cas particuliers, ou dédoubler certains cas trop généraux, mais la plupart restant inchangés.

Dans quelles conditions peut-on spécialiser un résultat général, sans refaire une preuve complète ?

Extension et raffinement

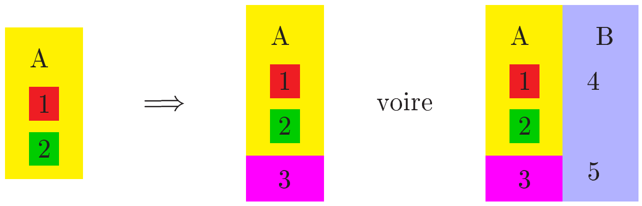
À partir d'un composant, en créer de nouveaux par :

– extension (verticale), *i.e.* ajout de fonctionnalités ;

– modification (verticale) ; ou

– adjonction (horizontale), *i.e.* raffinement de la structure de donnée.

Slide 11



Questions

(a) Est-ce que 1 et 2 peuvent être récursivement définis ?

(b) Est-ce que 3 peut remplacer ou modifier 2 ?

(c) En combinant (a) et (b) est-ce que 1 dépend de 3 ?

Difficultés de l'extension

Slide 12

Un couper/coller n'est pas une extension (pas de partage — par exemple, il faut refaire toute la preuve).

En mathématiques, on peut laisser l'utilisateur reconstruire des morceaux de preuve, mais on ne peut pas laisser le compilateur réécrire des morceaux de programme.

Difficulté supplémentaire dans le cas d'une preuve par récurrence (bon fondement).

Dans quelle mesure peut-on encore traiter les composants comme des boîtes noires (les compiler séparément) ?

Organisation modulaire

Slide 13

Quelque soit le domaine (industrie, mathématiques, informatique), on est amené à :

– Construire des composants élémentaires,
– Combiner ces composants.
– Utiliser une structure pyramidale : les composants sont les éléments de composants plus complexes.

En général, certaines parties peuvent être réutilisées inchangées, mais d'autres doivent être adaptées ou remplacées.

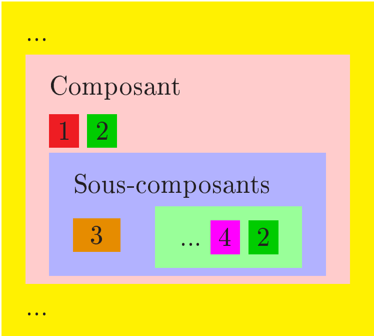
Décomposer le tout en parties peu indépendantes permet de réutiliser celles-ci telles quelles.

Une preuve qui utilise des lemmes auxiliaires difficiles peut elle-même être simple : la preuve peut alors facilement être refaite si les lemmes eux restent inchangés. Au pire, on ne retouchera que certains lemmes.

Organisation modulaire

Slide 14

Avec partage de sous-composantes, en profondeur (importance de la structure pyramidale) :



Voir par exemple `http://www-sor.inria.fr/~java/doc/javasoft/jdk/docs/api/tree.html` l'arbre des classes du langage Java ou le graphe de dépendance d'un système de module (ou d'une chaîne de fabrication d'airbus...).

La programmation modulaire

Slide 15

La complexité d'un programme :

- 100 lignes : script, un seul fichier, mais utilisation intensive des librairies.
- 1000 lignes : petite application, quelques fichiers. La modularité facilite l'écriture et la maintenance.
- 10000 lignes : application sérieuse. La modularité permet de mieux organiser le code et le développement en parallèle de plusieurs parties, la réutilisation de certaines composantes...
- 100000 lignes et plus : système complexe, probablement développé par plusieurs personnes devant évolué dans le temps, nécessitant une architecture robuste bien conçue et probablement le développement de librairies spécialisées.

Comparer à la complexité d'une usine pétro-chimique...

Un enjeu : la sûreté

Slide 16

La programmation modulaire a besoin de plus de sûreté

Un composant peut être utilisé en dehors du contexte dans lequel il a été écrit, par d'autres personnes, etc. Il faut donc une rigueur accrue : spécification claire, robustesse, résistance aux cas pathologiques, (*exemple d'Ariane 5*).

La modularité améliore la sûreté (et la sécurité)

Les composants peuvent être développés et testés séparément.

L'utilisation de librairies générales ou spécialisées tend à réduire la taille du noyau, les parties non spécifiques à l'application deviennent des opérations fournies en librairies.

Cacher les détails (un composant est une boîte noire avec une interface) évite de casser accidentellement des invariants (les variables internes sont inaccessibles).

Par un jeu de cache-cache

Slide 17

Cacher les détails de l'implémentation ... *le plus tôt possible !*

Augmente la sécurité, la réutilisabilité.

Cacher les valeurs (gestion de l'espace des noms)

Une valeur interne devient invisible et inaccessible de l'extérieur.

Facile à mettre en œuvre, mais limité : il faut parfois communiquer des informations à d'autres modules (par exemple pour les analyser/trier) tout en souhaitant que l'autre module traite (une partie) de ces informations comme des boîtes noires.

Cacher les types (rendre les types abstraits)

Une valeur ne peut être analysée (détruite) ou reconstruite qu'en connaissant sa structure (son type).

Par exemple, le facteur distribue le courrier, mais le courrier est cacheté et le facteur ne peut pas l'ouvrir.

Méthode plus puissante, mais légèrement plus difficile à mettre en œuvre : types (partiellement) abstraits.

Slide 18

Interface des composants

Décrire ce qui est utilisé (importé) et fourni (exporté) par un composant.

Exemple des prises électriques :
– Prise 20A *v.s.* 5A.
– Prise mâle (entrée) *v.s.* femelle (sortie).
– Une prise mâle avec terre ne peut pas être branchée dans une prise femelle sans terre.
– L'inverse est possible (sous-typage au niveau des interfaces)

Le typage est utilisé pour forcer le respect des interfaces (vérification statique et syntaxique).

Les branchements sont effectués par le compilateur.

Slide 19

La modularité du pauvre

La modularité est d'abord un style de programmation

Les langages avancés offrent des mécanismes pour favoriser la programmation modulaire, mais ne la force pas.

Inversement, des langages comme C qui offrent peu de mécanisme spécifiques, permettent quand même à l'utilisateur d'écrire des gros systèmes, mais avec beaucoup de discipline.

Slide 20

Se rappeler les premiers tuyaux du tronc commun, par exemple...
- Donner des noms aux constantes numériques.
- Partager le code en utilisant des fonctions auxiliaires. Par exemple, remplacer `C[A] ; ... ; C[B]` par

  ```
  c(x) { C[x] } c(A); ...; c(B)
  ```

  en C, ou par `let c(x) = C[x] in c(A) ... c(B)` en ML.

Utilisation de types abstraits

Il est important de cacher la représentation des données de telles façon que celle-ci puisse être changée sans affecter le reste du programme.

Pour cela on fournit des constructeurs (fonctions de création) et des destructeurs (fonctions d'accès) pour écrire et lire de telles données.

Slide 21

Le reste du programme doit être indépendant de la représentation choisie. Si le langage le permet, la représentation sera cachée par le typage dans le reste du programme (vérification automatique), sinon l'utilisateur doit être suffisamment discipliné pour respecter cette convention lui-même.

Exemple de type abstrait en C

```
typedef struct personne { char* nom; int age; } personne;
personne naissance (char* n, int a)
  { personne p; p.nom = n; p.age = a; return p; }
int age (personne p) { return p.age; }
char* nom (personne p) { return p.nom; }
```

Le reste du programme n'utilise que les fonctions `naissance`, `nom`, ... `age`.

Slide 22

La représentation peut alors être modifiée ou enrichie de façon transparente, par exemple en ajoutant un champ `sexe` :

```
typedef strut personne
  { char* nom, long date_naissance, short sexe } personne
int age (personne p) { return date() - r.date_naissance; }
```

Les outils du pauvre

Paramétrisation

En général, le seul outil pour l'abstraction est la fonction.

Le regroupement des paramètres dont dépend une application dans une structure (enregistrement) permet d'écrire le corps du programme comme une fonction de cette structure qui peut être appliqué avec des conditions initiales différentes.

Slide 23

Extensibilité

Traditionnellement (en C, Pascal, Caml, etc.), un programme est organisé autour des données :

1. Définition du type de donnée.
2. Écriture de fonctions pour manipuler les données.

L'ajout de nouvelles opérations sur les données est facile, mais le raffinement des données est difficile sans réécrire l'ensemble.

Exemple d'extension (verticale)

Slide 24

Un programme de la forme suivante

```
(* Définition des données *)
type forêt = Pinède | Chêneraie
(* Opérations sur les données *)
let dessiner_une_forêt = ...
let déboiser = Chêneraie -> couper | Pinède -> brûler
```

peut être étendu avec de nouvelles opérations

```
let reboiser = ...
```

sans changer le code précédent (l'extension peut même souvent se faire dans un fichier séparé).

Par contre ajouter un nouveau type de forêt, `Vierge` par exemple, oblige à réécrire toutes les fonctions.

Le problème de la pauvreté

Slide 25

Les solutions ci-dessus permettent de faire au mieux en l'absence de constructions modulaires spécialisés, mais elles sont limitées :

- Les fonctions permettent de paramétrer par rapport aux valeurs, mais pas par rapport aux types.
- L'application de fonction est une opération dynamique de bas niveau, alors que l'assemblage de composants pourrait se faire pendant le chargement (link). Ce sont des opérations plus macroscopiques.
- L'utilisation de type abstraits sans support dans le langage permet de programmer de façon modulaire, sans bénéficier de la sécurité offerte par un mécanisme de typage.

Il est important d'avoir des constructions primitives pour structurer le code.

Deux outils complémentaires

Les systèmes de modules

La notion de module est prise au sérieux.
- langage d'assemblage de structures basé sur l'abstraction.
- n'aide pas à l'extensibilité, n'aime pas la récursion.

Slide 26

Les langages à objets (et à classes)

La notion d'extensibilité est prise au sérieux.
- horizontalement par construction (ajout de classes), verticalement par l'héritage.
- l'abstraction de valeur est obtenu par masquage de certaines composantes.
- c'est un style de programmation très rigide, qui a ses terrains de prédilection, mais aussi ses limites. En particulier, la paramétrisation est plus difficile.

Les systèmes de modules

Système stratifié (couche au dessus du langage de base).

Une *structure* regroupe un ensemble de phrases du langage de base (déclaration de types, de valeurs, etc.)

Une *signature* décrit les composantes (types, valeurs, autres structures) publiques implémentées par une structure.

Slide 27

Une structure peut être paramétrée/instantiée par une autre structure (foncteur).

Un langage d'assemblage de structures, simple et puissant, avec une sémantique claire, mais pas (encore) :
- de structures récursives.
- d'extension, donc pas de réutilisation a posteriori.

L'approche à objets et à classes

Approche mixte (objets et classes)

Modularité en forme de peigne (schéma fréquent) :

$$\begin{tikzcd}
C_1 \arrow[r, Rightarrow] \arrow[d, dash] & C_2 \arrow[r, Rightarrow] \arrow[d, dash] & C_3 \arrow[r, Rightarrow] \arrow[d, dash] & C_4 \arrow[d, dash] \\
O_1 & O_2 & O_3 & O_4
\end{tikzcd}$$

Slide 28 Les classes sont extensibles mais ne sont pas des valeurs ordinaires. Les classes sont des modèles d'objets.

Les objets sont des valeurs, dérivées des classes (par instantiation), mais ne sont plus extensibles.

La sémantique est compliquée : l'extension d'une classe peut redéfinir d'anciennes méthodes, ce qui change le graphe d'appels récursifs (mécanisme dit de la *liaison tardive*)

Sous-typage

Modules et objets ont tous les deux une notion de sous-typage correspondant à la possibilité d'oublier des composantes (*c.f.* prises électriques).

On peut restreindre la signature d'une structure en cachant des composantes, ou le type d'un objet en ignorant certaines méthodes (en général le type d'un objet d'une sous-classe est un sous-type du type des
Slide 29 objets de la classe parente).

La notion de sous-typage est une approximation de la réalité.

Ses faiblesses sont compensées par des coercions *i.e.* par du typage *dynamique*) : par analogie électrique, un fil porte une couleur qui indique la phase, le neutre... Si un fil est décoloré, il faut tester pour retrouver lequel est la phase, et un échec est possible (ce qui est encore préférable à un court-circuit ou une erreur fatale).

Sous-typage : trouvez l'erreur !

Slide 30

```
class Point {
  private int x = 0;
  Point (int x0) { x = x0; }
  int getx () { return x; }
  Point max (Point p) {
    if (p.getx() > getx()) return p; else return this;
  }
}
class Bipoint extends Point {
  private int y = 0;
  Bipoint (int x0, int y0) { super (x0); y = y0; }
  int gety () { return y; }
  Bipoint max (Bipoint p) {
    if (p.gety() > gety()) return p; else return super.max (p);
  }
}
```

Les ingrédients d'un exemple complexe

Slide 31

Une armée modèle est composée d'une classe "soldat" et d'une classe "général". La communication est bi-directionnelle.

- Le soldat rend compte au général en s'identifiant (nom ou matricule) et en faisant un compte-rendu.
- Le général ordonne ou punit (le soldat).

Une instance plus informatique de ce protocole de communication se retrouve dans un window-manager : les fenêtres signalent des événements ; le manager ainsi informé a une vue globale de la disposition des fenêtres et de la souris et décide des actions à effectuer par exemple demander le rafraîchissement de certaines fenêtres.

Mots Clés

Exemple de Motif. Ici, chaque classe est paramétrée par rapport aux objets de l'autre classe pour rester extensible.

Les enjeux

Slide 32

Cet exemple illustre une configuration complexe dans laquelle des objets de plusieurs classes interagissent récursivement. Ces classes peuvent être regroupées dans une structure.

À partir de cette configuration modèle, il faut pouvoir :
- en prendre des instances.
- enrichir le modèle en raffinant le comportement de chaque classe, éventuellement en ajoutant des possibilités de communication entre les objets de chaque classe.
- enrichir le modèle en ajoutant de nouvelles classes interagissant avec les classes précédentes.

Bien sûr, nous exigeons une contrainte forte de sécurité où le typage seul doit garantir que tous les messages sont bien reçus (ce qui exclut l'utilisation du typage dynamique).

Extension du modèle

Slide 33

La configuration précédente est une *armée modèle*. Pour en faire une *armée de combat* on augmente la force de chaque intervenant tout en préservant le protocole de communication. Par exemple, on donne :
- deux étoiles au général pour augmenter son d'autorité.
- un fusil au soldat pour augmenter son d'agressivité.

Le général peut maintenant donner l'ordre de tirer au soldat.

Éventuellement, on étendra la configuration en introduisant de nouvelles classes, par exemple si un politique interagit avec le général, tout en préservant la correction du protocole précédent.

Mots clés Chaque classe hérite de la classe correspondante. Il est nécessaire de bien traiter les méthodes binaires pour préserver le typage.

Généralisation, reproduction en série

D'autres extensions sont naturellement possibles par exemple un soldat peut rapporter à deux généraux, même si elles mènent vite à l'anarchie...

Un pays peut aussi essayer d'exporter son modèle militaire (après avoir essayé plusieurs protocoles de communication).

Slide 34

Il est alors facile d'abstraire le modèle par rapport aux caractérisques propres aux soldats de chaque pays (sexe, âge, courage,...) tout en préservant certains secrets.

Mots clés

Un foncteur englobant réalise l'abstraction. L'utilisation d'une signature abstraite permet de bien préserver les secrets... et d'empêcher les soldats d'obéir au général d'une autre armée.

1

Les modules en Ocaml.

Slide 1

Cours	Exercices
1. Modules de base 2. Signatures 3. Restriction de signature 4. Compilation séparée 5. Foncteurs 6. Exemple avancé	1. Abstraction tardive... 2. Utilisation des librairies 3. Création d'une librairie 4. Polynômes

En quelques mots

Slide 2

- Un petit langage fonctionnel typé.
- Manipule des collections de définitions (de valeurs, de types) du langage de base.
- Leurs types : des collections de déclarations/spécifications (types des valeurs, déclarations des types).
- Modules emboîtés.
- Fonctions (foncteurs) et application de fonctions.

Largement indépendant du langage de base.

Modules de base

Slide 3

Les *structures* sont des collections de définitions.

`struct` $p_1 \ldots p_n$ `end`

Les définitions correspondent aux phrases du langage de base :

définitions de valeurs et de fonctions	`let` $x = \ldots$
définitions de types	`type` $t = \ldots$
définitions d'exceptions	`exception` E [`of` ...]
définitions de classes	`class` $C = \ldots$ `end`

Plus :

définition de sous-modules	`module` $X = \ldots$
définition de type de module	`module type` $S = \ldots$

Le nommage d'un module se fait à l'aide de la liaison `module`.

```
module S = struct type t = int let x = 0 let f x = x+1 end;;
```

Utilisation d'un module

On fait référence aux composantes d'un module avec la notation pointée *module.composante*

```
... (Z.f Z.x : Z.t) ...
```

Autre possibilité : la directive `open` *module* permet d'omettre le préfixe et le point :

Slide 4

```
open Z
... (f x : t) ...
```

Modules emboîtés

Un module peut être composante d'un autre module :

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end;;
```

Slide 5

La notation pointée et la construction `open` s'étendent naturellement aux sous-modules :

```
module Q =
  struct
    let z = T.R.x
    open T.R
      ...
  end
```

NB : La notation `open T.R` rend les composantes de `T.R` visibles dans la suite du corps de `Q` mais n'ajoute pas ces composantes à `Q`.

Les types des modules de base

Les *signatures* : collections de spécifications (de types).

Slide 6

`sig` s_1 s_2 … `end`

spécification de valeurs	`val x :` σ
spécification de types abstraits	`type` t
spécification de types manifestes	`type` $t = \tau$
spécification d'exceptions	`exception` E
spécification de sous-modules	`module` $X : M$
spécification de type de module	`module type` $S\ [= M]$

Nommage d'un type de module : par la liaison `module type`

```
module type MA_SIGNATURE = sig ... end
```

Synthèse de signature

Le système infère les signatures des modules (comme il infère les types des valeurs).

Slide 7

Module

```
module Exemple =
  struct
    type t = int
    module R =
      struct
        let x = 0
      end
    let y = R.x + 1
  end;;
```

Module

Signature inférée

```
module Exemple :
  sig
    type t = int
    module R :
      sig
        val x : int
      end
    val y : int
  end
```

Signature inférée

Restriction par une signature

Slide 8

La construction (*structure* : *signature*)

– vérifie que la structure satisfait la signature
  (toutes les composantes spécifiées dans la signature doivent être définies dans la structure, avec des types au moins aussi généraux);
– rend inaccessibles les parties de la structure qui ne sont pas spécifiées dans la signature;
– produit un résultat qui peut être lié par `module M = ...`

Sucre syntaxique :

```
module X : S = M
```

est équivalent à

```
module X = (M : S)
```

Restriction (Écritures équivalentes)

Slide 9

```
module M =
  (struct
     type t = int
     let x = 1
     let f y = x + y
   end :
   sig
     type t
     val f : int -> t
   end)
;;
```

```
module type S =
   sig
     type t
     val f : int -> t
   end
module M : S =
  struct
     type t = int
     let x = 1
     let f y = y + x
   end;;
```

```
M.x;;           (* ``Unbound value M.x'' *)
M.f 1 + 1;;     (* ``S.f 1 of type M.t is used with type int'' *)
```

Vues isomorphes incompatibles

Il est parfois important de distinguer des types isomorphes. Par exemples Euros et Dollars sont tous deux représentés par des flottants. Pourtant, il ne faut pas les confondre.

Ce sont deux espaces vectoriels, isomorphes mais disjoints, avec pour unités respectives l'`euro` et le `dollar`.

Slide 10

```
module Float =
  struct
    type t = float
    let un = 1.0
    let plus = (+.)
    let prod = ( *. )
  end;;
```

```
module type MONNAIE =
  sig
    type t
    val un : t
    val plus : t -> t -> t
    val prod : float -> t -> t
  end;;
```

La multiplication devient une opération externe sur les flottants.

Monnaies incompatibles

Dans `Float` le type `t` est concret donc il peut être confondu avec "float". Par contre, il est abstrait dans les modules `Euro` et `Dollar` ci-dessous :

```
module Euro = (Float : MONNAIE);;
module Dollar = (Float : MONNAIE);;
```

Les types `Euro.t` et `Dollar.t` sont isomorphes mais incompatibles.

Slide 11

```
let euro x = Euro.prod x Euro.un;;
Euro.plus (euro 10.0) (euro 20.0);;
Euro.plus (euro 50.0) Dollar.un;;
```

Pas de duplication de code entre `Euro` et `Dollar`.

Exercice

Exercice 1 *On voudrait implémenter une bureau de change. Pour simplifier, nous considérons seulement deux monnaies* `Euro` *et* `Dollar`*.*

Quel est le problème ? Avez-vous une idée de la solution ? Réponse

Donner la signature `BUREAU_DE_CHANGE` *des valeurs exportées par le module* `Bureau_de_change`*.* Réponse

Donner l'implémentation du module `Bureau_de_change`*.* Réponse

Slide 12 *Tester votre implémentation.* Réponse □

Vues multiples d'un même module

On peut donner une interface restreinte pour, dans un certain contexte, ne permettre que certaines opérations (typiquement, interdire la création de valeurs) :

Slide 13

```
module type PLUS =
  sig
    type t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS)
```

```
module type PLUS_Euro =
  sig
    type t = Euro.t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS_Euro)
```

À gauche, le type `Plus.t` est incompatible avec `Euro.t`.

À droite, le type `t` est partiellement abstrait et compatible avec "Euro.t", et la vue `Plus` permet de manipuler les valeurs construites avec la vue `Euro`.

La notation `with`

La notation `with` permet d'ajouter des égalités de types dans une signature existante.

L'expression `PLUS with type t = Euro.t` est une abréviation pour la signature

```
sig
  type t = Euro.t
  val plus: t -> t -> t
end
```

Slide 14

On peut alors écrire

```
module Plus = (Euro : PLUS with type t = Euro.t);;
Plus.plus Euro.un Euro.un;;
```

Elle permet de créer facilement des signatures partiellement abstraites.

Modules et compilation séparée

Une unité de compilation A se compose de deux fichiers :

- Le fichier d'implémentation `a.ml` :
  une suite de phrases
  semblable à l'intérieur de `struct ... end`
- Le fichier d'interface `a.mli` (optionnel) :
  une suite de spécifications
  semblable à l'intérieur de `sig ... end`

Slide 15

Une autre unité de compilation `B` peut faire référence à `A` comme si c'était une structure, en utilisant la notation pointée `A.x` ou bien en faisant `open A`.

Une fois les interfaces écrites et vérifiées (par le typeur) les composantes peuvent être développées et testées indépendamment.

La modularité du développement coïncide avec la modularité dans le langage.

Compilation séparée d'un programme

Slide 16

Fichiers sources : `a.ml`, `a.mli`, `b.ml`

Étapes de compilation :

`ocamlc -c a.mli`	compile l'interface de `A`	crée `a.cmi`
`ocamlc -c a.ml`	compile l'implémentation de `A`	crée `a.cmo`
`ocamlc -c b.ml`	compile l'implémentation de `B`	crée `b.cmo`
`ocamlc -o monprog a.cmo b.cmo`	édition de liens finale	

Les lignes 2 et 3 peuvent être échangées.

Le programme se comporte comme le code monolithique :

```
module A sig (* contenu de a.mli *) end =
   struct (* contenu de a.ml *) end
module B =
   struct (* contenu de b.ml *) end
```

L'ordre des définitions de modules correspond à l'ordre des fichiers objets `.cmo` sur la ligne de commande de l'éditeur de liens.

Utilisation de la commande `make`

Slide 17

Modules paramétrés

Un *foncteur* est une fonction des modules dans les modules :

`functor` $(Z : T)$ `->` M

Le *module* (corps du foncteur) est explicitement paramétré par le paramètre de module S. Il fait référence aux composantes de son paramètre avec la notation pointée.

Slide 18

```
module F = functor(Z : S) ->
  struct
    type u = Z.t * Z.t
    let y = Z.f 0
  end;;
```

Application de foncteur

On ne peut pas directement accéder dans `T`. Il faut au préalable l'appliquer explicitement à une implémentation de la signature `SIG` (tout comme on applique une fonction ordinaire).

```
module P1 = F(M1)
module P2 = F(M2)
```

`P1`, `P2` s'utilisent alors comme des structures ordinaires :

Slide 19

```
(P1.y : P2.u)
```

`P1` et `P2` partagent entièrement leur code.

Exemple (long) d'un compte bancaire

Slide 20

```
module type BANQUE =         (* vue du banquier *)
  sig
    type t
    type monnaie
    val créer : unit -> t
    val dépôt : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end

module type CLIENT =         (* vue donnée au client *)
  sig
    type t
    type monnaie
    val dépôt : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end;;
```

Une modélisation simple de la banque

Slide 21

Sur le modèle des actions, le compte est donné au client... de façon abstraite bien sûr (sa représentation est cachée) afin que seule la banque puisse modifier le compte du client.

```
module Banque_au_porteur (M : MONNAIE) :
    BANQUE with type monnaie = M.t =
  struct
    type monnaie = M.t and t = { mutable solde : monnaie }
    let zéro = M.prod 0.0 M.un and neg = M.prod (-1.0)
    let créer() = { solde = zéro }
    let dépôt c x =
      if x > zéro then c.solde <- M.plus c.solde x; c.solde
    let retrait c x =
      if c.solde > x then (c.solde <- M.plus c.solde (neg x); x) else zéro
  end;;
module Poste = Banque_au_porteur (Euro);;
```

La modularité dans l'exemple

– Les clients et le banquier ont des vues différents, mais compatibles, de la banque.

```
module Client : CLIENT
     with type monnaie = Poste.monnaie
     with type t = Poste.t
   = Poste;;
let mon_ccp = Poste.créer ();;
Poste.dépôt mon_ccp (euro 100.0);;
Client.dépôt mon_ccp (euro 100.0);;
```

Slide 22

– On peut créer des comptes dans différentes devises sans risque de confusion (elles seront détectées par le typage).

```
module Citybank = Banque_au_porteur (Dollar);;
let mon_compte_aux_US = Citybank.créer();;
Citybank.dépôt mon_ccp;;
Citybank.dépôt mon_compte_aux_US (euro 100.0);;
```

Modularité (suite)

– On peut changer l'implémentation de la banque tout en en préservant l'interface.

Pour éviter la fraude, une banque donne seulement au client un numéro de compte et conserve l'état de son compte en interne.

La représentation d'un compte devient un simple numéro.

Slide 23

– Plusieurs banques européennes ont alors des banques de données indépendantes (protégées).

```
module Banque_centrale = Banque_au_porteur (Euro);;
module Poste = Banque_au_porteur (Euro);;
```

Slide 24

```
module Banque (M : MONNAIE) : BANQUE with type monnaie = M.t =
  struct
    let zéro = M.prod 0.0 M.un and neg = M.prod (-1.0)
    type t = int
    type monnaie = M.t

    type compte = { numéro : int; mutable solde : monnaie }
    (* base de donnée de la banque *)
    let comptes = Hashtbl.create 10 and last = ref 0
    let compte n = Hashtbl.find comptes n

    let créer() = let n = incr last; !last in
      Hashtbl.add comptes n {numéro = n; solde = zéro}; n

    let dépôt n x = let c = compte n in
      if x > zéro then c.solde <- M.plus c.solde x; c.solde

    let retrait n x = let c = compte n in





      if c.solde > x then (c.solde <- M.plus c.solde x; x) else zéro
  end;;
```

Slide 25

Indépendance vis-à-vis de la représentation

`Banque_au_porteur` et "Banque" implémentent la même interface et sont interchangeables

```
module Poste = Banque_au_porteur(Euro);;
module La_Poste = Banque(Euro);;
module Citybank = Banque(Dollar);;
```

Slide 26 Il se trouve qu'on ne peut pas non plus observer la différence de comportement au niveau du langage, mais cela n'est pas garanti par le typage...

Utilisation des librairies

Exercice 2 ((*) Utilisation des librairies) Les amis des amis sont nos amis *revient à dire que la relation d'amitié est la fermeture transitive de la relation de grande amitié (amitié directe).*

Pour l'exercice on impose de représenter les personnes par leur nom (chaîne de caractères), les amis par des ensembles de personnes (librairie `ocaml/htmlman/manual058.html` *Set), et la relation de grande amitié par une table d'association (librairie* `ocaml/htmlman/manual043.html` *Hashtbl).*

Slide 27

On considère donné une certain nombre de personnes, et pour chaque personne l'ensemble de ses grands amis. Écrire une fonction qui calcule l'ensemble de tous les amis d'une personne et tester sur un petit exemple. Réponse □

Fabrication d'un librairie

Exercice 3 ((*) Les piles) *Définir la structure* `'a pile` *des piles d'élément de type* `'a`*, modifiables en place . Définir une fonction de* `créer: unit -> 'a pile` *qui retourne une pile vide. Écrire les fonctions* `ajouter : 'a -> 'a pile -> unit` *et* `retirer : 'a pile -> 'a` *d'ajout en pile et de retrait du dernier élément. On lancera une exception lorsque l'on essaye de retirer un élément d'une pile vide.* Réponse

Slide 28

En faire un module `Pile` *permettant de rendre la représentation des piles abstraite.* Réponse

Utiliser la commande make pour compiler le programme.

On voudrait maintenant écrire une version plus riche de `cpile`*, avec une fonction* `consulter` *qui retourne le sommet de la pile sans le retirer, mais on ne dispose que du code compiler et de son interface.* Réponse

Que manque-t-il pour fournir une implémentation plus efficace de la méthode sommet ?

Modifier la librairie pile pour en avoir deux vues différentes : (1) une vue concrète, extensibles, pour l'expert ; (2) une vue abstraite pour l'utilisateur ordinaire.

Ici, on pourra écrire tous les modules et leur types dans un seul fichier `piles.ml` *sans se soucier de la compilation séparée.* Réponse □

Slide 29

Polynômes

Exercice 4 (Polynômes à une variable (***))

- *Réaliser une bibliothèque fournissant les opérations sur les polynômes à une variable. Les coefficients forment un anneau passé en argument à la bibliothèque.* Réponse

Slide 30

- *Vérifier l'égalité* $(X+Y)(X-Y) = (X^2 - Y^2)$ *en considérant les polynômes à deux variables comme polynôme en* X *à coefficients les polynôme en* Y*.*
- *Vérifiez les mêmes égalités dans l'anneau Z/2Z. Z/2Z.* Réponse

Exercices (suite)

- *Écrire un programme qui prend un polynôme sur la ligne de commande et l'évalue en chacun des points lus dans* `stdin` *(un entier par ligne). Le résultat est imprimé dans* `stdout`*.* Réponse
- *Utiliser la commande make pour compiler le programme.* Réponse

Slide 31

- *Écrire une autre implémentation des polynômes, par exemple par des polynomes creux.*
- *Écrire une version spécialisée des polynômes dans Z/2Z utilisation une représentation pleine et des opérations logiques sur des vecteurs de bits (on pourra se limiter à des polynômes de faible degré à condition de tester le débordemnent).*

□

2

Classes et aux objets en Ocaml.

Slide 1

Cours	**Exercices**
1. Objets 2. Classes 3. Héritage 4. Typage 5. Abréviations 6. Sous-typage 7. Classes paramétrées 8. Méthodes virtuelles 9. Objets fonctionnels 10. Méthodes binaires	1. Classes 2. Les piles en style objet 3. Les chaînes en style objet 4. Variations sur la sauvegarde 5. Renommage

Avertissement

Slide 2

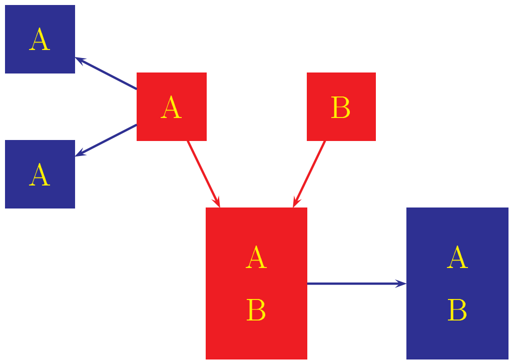
Ces notes introduisent informellement la couche objet du langage Caml. Le but n'est pas ici d'expliquer ce que sont les objets ou les classes, mais d'apprendre à les utiliser d'abord de façon intuitive dans des situations simples.

Nous présenterons également des exemples plus avancées d'utilisation les objets et des classes et nous expliquerons plus formellement le concept d'objet dans les cours suivants.

Objets et classes

Slide 3

Les classes (en rouge) sont des modèles d'objets, extensibles (les flèchent rouge représente l'héritage).



Les objets (en bleu) sont des instances (flèches bleu) des classes, rigides. Ce sont des valeurs qui peuvent être arguments ou résultats.

Les objets

Un objet est un enregistrement qui regroupe deux sortes de champs :

– Les **variables d'instance**, éventuellement mutables, forment l'état interne de l'objet ; ce sont des valeurs qui ne sont accessibles que par les méthodes.
– Les **méthodes** sont des fonctions qui ont accès en plus de leurs arguments aux variables d'instance et à l'objet lui-même.

Slide 4 Les objets sont construits à partir des classes.

La seule opération possible sur les objets depuis l'extérieur (*i.e.* en dehors de la classe qui le définit) est l'appel de méthode aussi appelé envoi de message.

objet#méthode [arguments]

Exemple : un compte en banque

État interne	Méthodes
– Solde du compte,	– Consulter le solde,
– Opérations récentes,	– Déposer de l'argent,
– Découvert autorisé,	– Retirer de l'argent,
	– Imprimer un relevé.

Slide 5

Exemple de messages envoyés à un objet compte `c` :

```
c # dépôt (100.0)
c # retrait (50.0)
c # relevé
```

Les classes

Slide 6

Un mini langage de classes pour définir des fonctions et les appliquer à des arguments.

Les définitions de classes sont restreintes au toplevel.

Une classe totalement appliquée est un modèle d'objet (un *moule*) défini par :
– un ensemble de variables d'instances
– un ensemble de méthodes.
Au besoin, un moule est aussi abstrait, explicitement, par rapport à l'objet (souvent appelé self) qui sera dans le moule au moment de l'exécution des méthodes

On crée une instance du modèle par `new` *moule*

Exemples

Slide 7

```
class vide = object end;;
let objet_vide = new vide;;
let un_autre_objet_vide = new vide;;
un_autre_objet_vide = objet_vide;;
```

– : bool = false

```
class élève son_nom = object
  val nom : string = son_nom
  val mutable moyenne = 0.
  val mutable nombre_de_notes = 0
  method note x =
    let n = float nombre_de_notes in
    moyenne <- (x +. moyenne *. n) /. (n +. 1.);
    nombre_de_notes <- nombre_de_notes + 1; moyenne
  method bon_élève = moyenne > 14.
end;;
```

Exercice

Exercice 5 (Exemple (*)) *Transformer un petit programme simple, monomorphe, sans classe en un programme avec une seule classe la plus englobante possible. Y a-t-il un intérêt à cette transformation ?* Réponse

Slide 8

Essayer de transformer une librairie de fonctions polymorphes, par exemple sur les listes. Quel est le problème rencontré ? Réponse □

Syntaxe des classes (simplifiée)

Slide 9

Définition (de classe)

`class` [`virtual`] [`[` Type-var* `]`] Nom [Variable*] `=` Classe

Classe

Nom
`fun` Variable* `->` Classe
Classe Expression*
`object` [`(` Variable [`:` Type] `)`] Corps `end`

Corps (de classe)

[Type*] `inherit` Classe [`as` Variable]
`val` [`mutable`] Variable `=` Expression
`method` [`virtual`] [`private`] Variable `=` Expression
`initializer` Expression

Expressions (nouvelles)

Variable `<-` Expression

Expression # Variable
new Classe

Slide 10

Héritage : compte bancaire

```
class compte =
  object
    val mutable solde = 0.0
    method solde = solde
    method dépôt x = solde <- solde +. x
    method retrait x =
      if x <= solde then (solde <- solde -. x; x) else 0.0
  end;;
let c = new compte in c # dépôt 100.0; c # retrait 50.0;;
```

Slide 11

Envoi de message à soi-même.

```
class compte_avec_intérêts =
  object (self)
    inherit compte
    method intérêts = self # dépôt (0.03 *. self # solde)
  end;;
```

Méthode privée

Il peut être utile de ne pas donner la méthode `intérêts` à l'utilisateur, mais seulement de la rendre visible dans les sous-classes en vue d'une utilisation ultérieure.

Slide 12

On en fait une méthode privée :

```
class compte_avec_intérêts =
  object (self)
    inherit compte
    method private intérêts =
      self # dépôt (0.03 *. self # solde)
  end;;
```

Les méthodes privées ne sont pas accessibles de l'extérieur

```
(new compte_avec_intérêts) # intérêts
```

Liaison tardive

Un appel récursif à une méthode prend en compte sa redéfinition ultérieure ; c'est le mécanisme de la liaison tardive.

Slide 13

Par exemple si on corrige le comportement de la méthode dépôt pour éviter une usage abusif :

```
class bon_compte =
  object
    inherit compte_avec_intérêts
    method dépôt x = if x > 0.0 then solde <- solde +. x
  end;;
```

La méthode `intérêts` appellera la nouvelle définition de la méthode `dépôt`.

Raffinement d'une méthode

Slide 14

Lors de l'héritage, on peut lier la vue de l'objet dans la classe parente pour appeler les anciennes méthodes.

Ainsi, on aurait pu définir :

```
class bon_compte =
  object
    inherit compte_avec_intérêts as super
    method dépôt x =
      if x > 0.0 then super # dépôt x
      else raise (Invalid_argument "dépôt")
  end;;
```

Ce qui n'oblige pas à connaître l'implémentation actuelle de la méthode `dépôt`

Compte avec relevé

Slide 15

Mélange des aspects précédents dans un compte avec relevé

```
type opération = Dépôt of float | Retrait of float

class compte_avec_relevé =
  object (self)
    inherit bon_compte as super
    val mutable registre = []
    method private trace x = registre <- x::registre
    method dépôt x =
      self#trace (Dépôt x); super # dépôt x
    method retrait x =
      self#trace (Retrait x); super # retrait x
    method relevé = List.rev registre
  end;;
```

Initialisation des objets

Slide 16

On peut abstraire une classe par rapport à des valeurs initiales.

Le mieux aurait été de le prévoir dès le départ. On peut aussi utiliser la clause `initializer` pour rattraper le coup.

Paramétrisation a priori

```
class compte x =
 object
   val mutable solde = x
   method ...
   ...
 end
```

Paramétrisation a posteriori

```
class promotion x =
 object (self)
   inherit compte_avec_relevé
   initializer solde <- x
 end;;
```

Les clauses `initializer` sont exécutées immédiatement après la création de l'objet, dans l'ordre de définition. Elles ont accès aux variables d'instance et à self.

Plus souvent utilisées pour assurer des invariants.

Clause d'initialisation

Slide 17

On peut aussi effectuer un dépôt automatique après la création de l'objet (ce qui correspond à l'idée de promotion).

```
class promotion x =
  object (self)
    inherit compte_avec_relevé
    initializer self # dépôt x
  end;;
```

Cette version effectue l'initialisation comme un dépôt, ce qui empêche automatiquement l'initialisation avec une valeur négative.

```
let ccp = new promotion 100.0 in
ignore (ccp # retrait 50.0); ccp # relevé;;
```

– : opération list = [Dépôt 100; Retrait 50]

Types des objets

Slide 18

Les types des objets sont de petits enregistrements de types :

Le type d'un compte bancaire :

```
type compte_bancaire =
  < solde : float;
    dépôt : float -> unit;
    retrait : float -> float;
    relevé : opération list >;;
(new promotion 50.0 : compte_bancaire);;
(new compte : compte_bancaire);;
```

Le type `compte_bancaire` est dit fermé (entièrement spécifié)

Types des objets ouverts

Slide 19

Le type d'un objet peut être partiellement spécifié.

```
let fermeture c = c # retrait (c # solde);;
```

val fermeture : < retrait : 'a -> 'b; solde : 'a; .. > -> 'b = <fun>

Cette fonction, polymorphe, peut s'appliquer à toutes sortes de comptes. Les `..` se comporte comme une variable de type anonyme.

```
fermeture (new compte);;
fermeture (new promotion 100.0);;
```

Le partage est exprimé et conservé par la construction `as`

```
let dépôt c x = if x > 100.0 then c # dépôt x; c;;
```

val dépôt : (< dépôt : float -> unit; .. > as 'a) -> float -> 'a = <fun>

```
(dépôt (new compte), dépôt (new compte_avec_relevé));;
```

– : (float -> compte) ∗ (float -> compte_avec_relevé) = <fun>, <fun>

Types des classes

Les types des classes sont des petites spécifications qui décrivent la structure des classes.

Dans les classes, le type de `self` est toujours ouvert (partiellement spécifié). Il est compatible avec l'ensemble des méthodes couramment définies.

Slide 20

Syntaxe des types des classes

Type-de-définition-de-classe

`class` [`virtual`] [`[` Type-var* `]`] Nom [Type*] `:` T

Définition-de-type-de-classe

`class type` [`virtual`] [`[` Type-var* `]`] Nom [Type*] `=` T

T (Type de classe)

Nom

Type `->` T

`object` [`(` Type `)`] Type-du-corps `end`

Slide 21

Type-du-corps

`val` [`mutable`] Variable `:` Type

`method` [`virtual`] [`private`] Variable `:` Type

Nouvelles expressions de type

`<` variable `:` Type [`;` variable `:` Type] * [`; ..`] `>`

`(` Type `as` Type-var `)`

Exemple

Le type (inféré) des classes avec relevé :

```
class compte_avec_relevé :
  object
    val mutable registre : opération list
    val mutable solde : float
    method dépôt : float -> unit
    method private intérêts : unit
    method relevé : opération list
    method retrait : float -> float
    method solde : float
    method private trace : opération -> unit
  end
```

Slide 22

Abréviations automatiques

Les définitions de classes (et de types de classes) créent automatiquement des abréviations de types

```
type compte_avec_relevé =
   < dépôt : float -> unit; relevé : opération list;
     retrait : float -> float; solde : float >
```

Slide 23

```
type #compte_avec_relevé =
   < dépôt : float -> unit; relevé : opération list;
     retrait : float -> float; solde : float; .. >
```

Le premier est une instance du second. En fait tout objet d'une sous-classe de `compte_avec_relevé` est une instance du second.

```
fun x -> ( (x : compte_avec_relevé) : #compte);;
```

Sous-typage sur les objets

Slide 24

Les types `compte_avec_relevé` et `compte` sont incompatibles.

```
fun x -> ( (x : compte_avec_relevé) : compte);;
```

Mais on peut explicitement coercer l'un vers l'autre :

```
fun x -> (x : compte_avec_relevé :> compte);;
```

Il existe un raccourci, moins puissant mais souvent suffisant *(utiliser la version longue lorsque le raccourci ne s'applique pas)*

```
fun x -> (x :> compte);;
```

Par exemple, pour mettre les comptes avec ou sans relevé dans une même collection :

```
[ (new compte :> compte);
  (new compte_avec_relevé :> compte) ];;
```

– : compte list = [<obj>; <obj>]

Règle de sous-typage (approximation)

Slide 25

Un type d'objet A est un sous-type d'un type d'objet B si A et B sont égaux ou bien tous deux fermés et tels que :

- chaque méthode de A est une méthode de B ;
- son type dans A est un sous-type de son type dans B (sous-typage en profondeur) ;

Le type de $D_A \to I_A$ d'une fonction est sous-type de $D_B \to I_B$ si

- I_A est sous-type de I_B *on peut agrandir l'***I***mage*
- D_B est sous-type de D_A *on peut rétrécir son* **D***omaine*

En particulier, si A et B contiennent une même *méthode binaire* (*c.f.* ci-après), par exemple

```
A = (< m : 'a -> unit; ... > as 'a)
B = (< m : 'b -> unit; ... > as 'b)
```

alors A est sous-type de B seulement s'il est égal à B. En effet, pour que A soit sous-type de B la contavariance sur la méthode m ne impose que B soit aussi sous-type de A, et la seule solution est que A soit égal à B.

(On vérifie facilement la relation de sous-typage est antisymmétrique.)

Slide 26

Slide 27

Que peut-on cacher dans une classe ?

Les variables d'instance et les méthodes privées peuvent être cachées en donnant à la classe une contrainte de type.

```
class foo : object method m : int end =
  object (self)
    val foo = 2
    method private bar = foo * foo
    method m = self # bar * self # bar
  end;;
```

Le même effet peut être produit par une contrainte de signature

```
module Truc : sig class inutile : object end end =
  struct
    class inutile = object val foo = 2 end
  end;;
```

Classes paramétrées

Slide 28

Une mappe est une table d'association.

```
class type ['a, 'b] mappe =
  object
    method trouve : 'a -> 'b
    method ajoute : 'a -> 'b -> unit
  end;;
```

Une implémentation simple et raisonnable pour de petites mappes avec des listes :

```
class ['a, 'b] petite_mappe : ['a, 'b] mappe =
  object
    val mutable table = []
    method trouve clé = List.assoc clé table
    method ajoute clé valeur = table <- (clé, valeur) :: table
  end;;
```

Classes paramétrées

Slide 29

Une implémentation plus efficace pour les mappes de grande taille avec une table de hache. Pour résoudre les conflits, les éléments de la table de hache sont eux mêmes des petites mappes.

```
class ['a, 'b] grande_mappe taille : ['a, 'b] mappe =
  object (self)
    val table =
      Array.init taille (fun i -> new petite_mappe)
    method private hash clé =
      (Hashtbl.hash clé) mod (Array.length table)
    method trouve clé = table.(self#hash clé) # trouve clé
    method ajoute clé = table.(self#hash clé) # ajoute clé
  end;;
```

Une version plus efficace pourrait automatiquement retailler la table de hache lorsque les petites mappes deviennent trop grandes (à condition de leur ajouter une méthode taille).

Exercice

Exercice 6 (Piles (*)) *Écrire une classe des piles (d'entiers)*
Généraliser en une classe paramétrique des piles. Réponse
Notez les différences avec les piles comme module. □

Slide 30

Classes et méthodes virtuelles

Une méthode est virtuelle lorsqu'elle est utilisée mais pas définie. La classe est alors elle même virtuelle : on peut en dériver des sous-classes, mais pas en créer des instances.

Les méthodes virtuelles peuvent être utilisées pour définir un comportement commun aux sous-classes utilisant des caractéristiques pas encore définies des sous-classes.

Une classe virtuelle doit être indiquée comme telle. (ce qui évite qu'une méthode mal orthographiée deviennent accidentellement virtuelle...)

Exemple Une classe définit des règles générales, un protocole, etc. Une ou plusieurs sous classes implémentent une stratégie, un comportement plus précis.

Slide 31

Exemple : jeu et stratégies

Slide 32

Les règles du jeu sont partagées

```
class virtual joueur nom =
  object (self : 'a)
    method virtual coup : int
    method vérifie n = 1 + (n-1) mod 2
    method joue_avec (partenaire : 'a) jeu =
      if  (jeu = 0) then (nom : string) else
      let suivant = jeu - (self # vérifie (self # coup)) in
      partenaire # joue_avec self suivant
  end;;
```

Exemple : jeu et stratégies

Slide 33

Deux stratégies indépendantes

```
class petit nom =
  object
    inherit joueur nom
    method coup = 1
  end;;
```

```
class hasard nom =
  object
    inherit joueur nom
    method coup =
      1 + Random.int 2
  end;;
```

La partie

```
(new hasard "Pierre") # joue_avec
  (new petit "Jacques") 26;;
```

Les modificateurs de champs

Les modificateurs de champs sont visibles et hérités dans les sous-classes, à moins qu'ils ne soient masqués par le typage.

val mutable

Ce modificateur peut être ajouté tardivement et masqué par le typage.

Slide 34

method private

Annotation à placer dès l'origine, héritée. La méthode, invisible dans l'objet, peut être cachée dans la classe par le typage.

class virtual, method virtual

Annotation obligatoire pour une méthode utilisée mais non définie et pour une classe comprenant une telle méthode. Propriété héritée qui disparaît après définition de la méthode.

Objets dans le style fonctionnel

On peut retourner une copie superficielle d'un objet quelconque en utilisant la fonction de librairie :

```
Oo.copy : (< .. > as 'a) -> 'a
```

Dans une définition de classe, on peut aussi retourner une copie de self avec certains champs éventuellement modifiés par la construction :

Slide 35

$$\{\langle \ell_1 = e_1; \ldots \ell_k = e_k \rangle\}$$

où les champs ℓ_1 à ℓ_k sont un sous-ensemble des variables d'instances.

Exemple : les mappes fonctionnelles

La méthode `ajoute` doit retourner une nouvelle mappe, sans modifier l'ancienne.

```
class type ['a, 'b] mappe_fonctionnelle =
  object ('mytype)
    method trouve : 'a -> 'b
    method ajoute : 'a -> 'b -> 'mytype
  end;;
```

Slide 36

Des petites mappes peuvent être représentées par des listes :

```
class ['a, 'b] petite_mappe : ['a, 'b] mappe_fonctionnelle =
  object
    val mutable table = []
    method trouve clé = List.assoc clé table
    method ajoute clé valeur =
      {< table = (clé, valeur) :: table >}
  end;;
```

Héritage multiple

Une classe en librairie...

```
class sauvegarde =
  object (self : 'mytype)
    val mutable copie = None
    method sauve = copie <- Some {< >}
    method récupère =
      match copie with (Some x) -> x | _ -> raise Not_found
  end;;
```

Slide 37

Note : ici on peut remplacer `{< >}` par `(Oo.copy self)`.

Les fonctionnalités de la classe `sauvegarde` peuvent être ajoutées *a posteriori* à (presque) n'importe quelle autre classe :

```
class compte_protégé =
  object
    inherit compte
    inherit sauvegarde
```

```
end;;
```

Slide 38

Exercices

Exercice 7 (Sauvegarde) *Les fonctionnalités de la classe sauvegarde peuvent être ajoutées à* presque *n'importe quelle autre classe. Pourquoi* presque *?* Réponse

Slide 39

La classe sauvergarde permet chaîne en fait les versions entre elles. On peut donc récupérer une version de deuxième génération entre récupérant la sauvegarde de la sauvegarde.

Écrire une variante de la classe sauvegarde qui ne conserve qu'une seule copie. Quel est l'intérêt de cette modification (si une seule copie est nécessaire) ? Réponse

Écrire une variante fonctionnelle `fsauvegarde` de la classe sauvegarde : la méthode récupère retourne un objet qui est une copie de `self` dans laquelle le champ `original` pointer vers la version originale. Il faut continuer avec la copie pour pouvoir plus tard récupérer la version originale. Réponse

On reprend la version qui conserve les sauvegardes intermédiaires. Rafiner la version impérative de la sauvergarde en profondeur en ajoutant une méthode `balai` *qui conserve les sauvegardes de façon logarithmique (*i.e. *ne conserve que les versions d'âge* 2^0, 2^1, ... 2^n*).* Réponse □

Slide 40

Slide 41

Méthodes binaires

Ce sont des méthodes qui prennent parmi leur argument un objet du même type que self.

Les vraies

```
class binaire = object (self)
  method choisir x = if Random.int 1 = 0 then x else self
end;;
class binaire : object ('a) method choisir : 'a -> 'a end
```

Les objets avec des méthodes binaires ne peuvent pas être coercés tant que les méthodes binaires restent visibles.

Les fausses La contrainte de type à l'origine de la méthode binaire peut être relâchée, en générale en rendant la classe paramétrique en le type de cet argument.

Méthodes binaires et classes paramétrées

Slide 42

```
class étoile =
  object (self : 'a)
    val mutable pos = 0
    method pos = pos
    method eq (x : 'a) =
      (x#pos = self#pos)
  end;;
```

```
class étoile : object ('a)
  val mutable pos : int
  method eq : 'a -> bool
  method pos : int
end
```

```
class ['a] étoile =
  object (self)
    val mutable pos = 0
    method pos = pos
    method eq (x : 'a) =
      (x#pos = self#pos)
  end;;
```

```
class ['a] étoile : object
  constraint 'a = < pos:int; .. >
  val mutable pos : int
  method eq : 'a -> bool
  method pos : int
end
```

Exercices

Slide 43

Exercice 8 (Méthodes binaires (**)) *Créer une classe* `string` *avec les fonctionnalités essentielles des chaînes.*

Ajouter une méthode `concat` *aux chaînes de caractères.*

Quel est le problème éventuel ? Réponse ☐

Exercices (suite)

Slide 44

Exercice 9 (Renommage (***)) *En considérant que la monnaie par défaut est le franc, étendre la classe* `compte` *pour fournir des dépôts et des retraits en euros, les conversions étant explicites. (Ici, on ne demande pas de cacher la représentation de la monnaie par l'utilisation de type abstrait ; on pourra donc utiliser des float à la fois pour représenter les dollars et les francs. Voir l'exercice sur les taux de changes.)*

Reprendre le même exercice, mais en définissant d'abord une classe `euro_compte` *puis une ensuite classe* `compte_mixte` *par héritage multiple.*

Quel est le problème éventuel ? Réponse □

3

Classes et modules.

Slide 1

Cours	Exercices
1. Point de rencontre 2. Groupement des classes 3. Constructeurs de classes 4. Abstraction et héritage 5. Classes amies 6. Combinaison intime	1. Structures de données (Listes, concaténation, cellules) 2. Bank 3. Ensembles 4. Polynomes

Résumé

Slide 2

Modules et classes remplissent des rôles différents.

- Les modules gèrent l'abstraction de type et de valeur et la paramétrisation par des types et des valeurs *en gros*.
- Les classes n'offrent pas d'abstraction de type, ni la possibilité de groupement.
- Elles offrent le mécanisme de liaison tardive (classes), l'héritage (classe) et bien sûr la programmation par envoi de message (objets).

Il est fréquent de combiner les deux mécanismes pour bénéficier de tous les atouts simultanément.

Point de rencontre des classes et des modules

Slide 3

Quelques exemples s'expriment aussi facilement dans le langage des classes que dans celui des modules.

L'intersection est à la fois grande et petite :

- De nombreux programmes peuvent être écrits dans un style ou dans l'autre.
- Cependant, la modularité n'est pas toujours la même selon le style utilisé.
- Pour des exemples peu modulaire et qui utilisent peu le mécanisme à objet, le choix du style n'est pas fondamentale... au point que ces exemples peuvent également être écrit dans le langage de base.

Point de rencontre : types abstraits algébriques...

L'exemple le plus typique est de point de rencontre est l'implémentation d'un type abstrait algébrique, ... avec modification en place et des opérations unaires !

Slide 4

```
class ['a] stack = object

  val mutable p : 'a list = []
  method push v = p <- v :: p
  method pop =
    match p with
    | h :: t -> p <- t; h
    | [] -> failwith "Empty"
end;;
```

```
module Stack : STACK = struct
  type 'a t = 'a list ref
  let create () = ref []
  let push x s = s := x :: !s
  let pop s =
    match !s with
      h::t -> s := t; h
    | [] -> failwith "Empty"
end;;
```

Résumé	**classe**	**module**
Le type des piles	`'a stack`	`'a Stack.t`
Création d'une pile	`new stack`	`Stack.create ()`
Empiler x sur s	s`#push` x	`Stack.push` x s
Dépiler s	s`#pop`	`Stack.pop` s

Slide 5

Extensibilité

Slide 6

```
class ['a] stack_ext =
  object (self)
    inherit ['a] stack




    method top =
      let s = self#pop in
      self#push s; s
end;;
```

```
module StackExt = struct
  (* include Stack *)
  type 'a t = 'a Stack.t
  let create = Stack.create
  let push = Stack.push
  let pop = Stack.pop
  let top p =
    let s = pop p in
    push s p; s
end;;
```

Cependant, les modules ne permettent pas la liaison tardive (important si les fonctions sont définies récursivement). Par exemple, la redéfinition de `pop` n'affectera pas le comportement de `top`.

Quelle approche choisir ?

Slide 7

C'est une affaire de goût quand les deux sont possibles.

Classes quand la liaison tardive est nécessaire.

Modules s'il y a des méthodes binaires or deux types abstraits liés, *e.g.* vecteurs et matrices (leurs représentations doivent être partagées).

Approche combinée dans de nombreux cas.
- utilisation orthogonale : chacun son rôle, indépendemment.
- combinaison intime : l'un complète l'autre.

Regroupement les classes liées

Slide 8

C'est une utilisation orthogonale de modules et des classes.

Les classes `nil` et `cons` sont liées, mais seulement par l'usage.

```
module Liste = struct
  exception Nil
  class ['a] nil =
    object (self : 'alist)
      method hd : 'a = raise Nil
      method tl : 'alist = raise Nil
      method null = true
    end
```

```
  class ['a] cons h t =
    object (_ : 'alist)
      val hd = h val tl = t
      method hd : 'a = h
      method tl : 'alist = t
      method null = false
    end
end;;
```

NOTE : En Ocaml, les listes sont plus naturellement représentées par un type somme qui permet en outre d'utiliser le filtrage.

Extensibilité de classes liées

Slide 9

Typiquement, les deux classes seront étendues simultanément (bien que cela ne soit pas obligatoire)

```
module Liste1 = struct
  class ['a] nil =
    object
      inherit ['a] Liste.nil
      method length = 0
    end
      ;;
```

```
class ['a] cons h t =
  object
    inherit ['a] Liste.cons h t
    method length =
      1+tl#length
  end
end;;
```

On peut aussi étendre les listes par l'ajout d'un nouveau constructeur `append`, ce qui revient à ajouter une nouvelle classe avec la même interface.

Voir exercice.

Exercice (structures de données)

Slide 10

Exercice 10 *Reprendre le module des listes en fournissant d'abord une classe virtuelle pour les différents types de listes.* Réponse

Implémenter une classe `append` *prenant deux listes et se comportant comme une liste.* Réponse

Montrer que les listes peuvent être définies à partir d'une classe `['a,'b] cellule` *implémentant une cellule à deux cases (non homogènes) et d'une classe* `['a, 'b] nulle` *ayant la même interface mais implémentant le pointeur vide. Retrouver les classes* `nil` *et* `cons` *comme des spécialisations des classes* `nulle` *et* `cellule`. Réponse □

Constructeurs de classes

Slide 11

En Ocaml, on ne peut créer un objet d'une classe que par la construction `new`. Pour obtenir plusieurs constructeurs de la même classe, on peut

- utiliser des fonctions auxiliaires de construction.

  Celles-ci ne peuvent pas être héritées.
- utiliser des classes auxiliaires.

  Ells peuvent être héritées, mais séparément.
- utiliser des wrappers ?

  Ils ne peuvent pas partager les variables d'instance.
- mettre les constructeurs dans des méthodes privées et utiliser les clauses `initializers` pour les sélectionner.
- utiliser un seul constructeur en utilisant des arguments optionnels nommés ou étiquettés.

Utilisation de fonctions auxiliaires

On définit une classe générale

Slide 12

```
class c x0 y0 =
  object (* class principale *)
    val x = x0
    val y = y0
    method peu_importe = if x then y else 0
  end;;
```

puis des fonctions de création d'objets de cette classe.

```
let c0 () = new c false 0
and c1 b = new c b 0
and c2 b x = new c (b && x > 0) (x * x);;
let p = c2 true 1 and q = c0();;
```

Ces constructeurs ne peuvent pas être hérités.

Utilisation de classes auxilliaires

On définit des variantes de la classe générale

Slide 13

```
class c0 = c false 0
class c1 b = c b 0
class c2 b z = c (b && z > 0) (z * z);;
let p = new c2 true 1 and q = new c0;;
```

Les classes utilisées comme constructeurs peuvent être héritées :

```
class c2' b x = object
  inherit c2 b x
  initializer Printf.printf "générale3 val y = %d\n" y
end;;
let _ = new c2' true 3;;
```

Ces constructeurs peuvent être hérités, mais individuellement, sans partage du code ajouté.

Utilisation d'un wrapper

Contrainte les fonctionnalités ajoutées n'accèdent pas aux variables d'instances...

Une classe et ses constructeurs

Slide 14

```
class wc = object
  method une_autre = Printf.printf "indépendante"
class c0' = object inherit c inherit une_autre end
class c1' x = object inherit c x inherit une_autre end
```

Situation idéale... mais assez rare

Utilisation de méthodes privées

On met les constructeurs dans la classe sous forme de méthodes privées qui seront hérités. On utiliser les initializer pour appeler tel ou tel constructeur. Cela fonctionne bien pour des champs mutables définis avec des valeurs par défaut.

Slide 15

```
class c = object
  val mutable x = 0
  method private c0 = x <- 10
  method private c1 x0 = x <- x0
end;;
class c0 = object (s) inherit c initializer s#c0 end
class c1 z = object (s) inherit c initializer s#c1 z end;;
```

On peut hériter des constructeurs,

mais pas changer leur type.

Slide 16

Méthodes privées avec héritage

```
class c' = object
  inherit c as s
  val mutable y = 0
  method private c0 = s#c0; y <- 100
  method private c1' x0 y0 = s#c1 x0; y <- y0
end;;
class c0 = object (s) inherit c' initializer s#c0 end
class c1 x y =
 object (s) inherit c' initializer s#c1' x y end;;
```

Constructeur unique

Slide 17

Le principe : On distingue les différents cas par un type somme.

```
type arg_c = Zéro | Un of bool | Deux of bool * int
class c args =
  let x0, y0 = match args with
    | Zéro -> false, 0
    | Un b -> b, 0
    | Deux (b,x) -> b, x in
  object
    val x = x0 val y = y0
    method peu_importe = if x then y else 0
  end ;;
```

Problème c'est très lourd...

Solution Arguments (nommés) optionnels ou étiquettés.

Class avec arguments étiquetés

Slide 18

Voir la construction correspondante de Ocaml.

Ils évitent la déclaration de types concrets éphémères :

```
class point arg =
  let x0 =
    match arg with `Int x -> float x | `Float x -> x in
  object
    val x = x0
    method getx = x
  end ;;
let p = new point (`Int 2);;
let q = new point (`Float 2.5);;
```

Classe avec arguments optionnels

Slide 19

Voir la construction correspondante de Ocaml.

```
class c ?b:(x0 = false) ?x:(y0 = 0) =
  object
    val x = x0 val y = y0
    method peu_importe = if x then y else 0
  end ;;
```

Héritage avec arguments optionnels

```
class c' ?b ?x = object
  inherit c ?b ?x
  initializer Printf.printf "new c' with val y = %d\n" y
end ;;
let p = new c and q = new c true 3;;
```

Exercice 11 (Constructeurs) *On reprend l'exemple de la banque mais en style objet style objet, afin de rester extensible. En particulier, on veut pouvoir raffiner le modèle de la banque tout en héritant des constructeurs* *(i.e. comment spécialiser la version de la banque et du client simultanément) ?*

La banque peut regrouper un ensemble de services financiers liés dans un même module `Compte`*.*

Définir une classe `compte_bancaire` *utilisée pour des extensions ultérieures. (On pourra reprendre, par exemple, la classe* `compte_avec_relevé` *Elle sera cachée dans la vue montrée au client.*

Slide 20 *Définir un type de classe* `VUE_DU_CLIENT`*.*

Définir une classe `client` *permettant au client de gérer ses comptes (sans pouvoir en créer) et si besoin d'hériter de cette classe pour personnaliser la gestion de son compte.*

Définir une fonction de construction `promotion` *permettant une offre promotionnelle, non modifiable, (qui ne pourra donc pas être héritée).* Réponse

Le client spécialise sa vision du compte client. Par exemple, founrnir une version dérivée de la classe client lui permettant de gérer lui-même ses comptes (indépendemment du relevé de la banque). Cette classe prend en paramètre un compte auxilliaire servant d'épargne et sur lequel sont effectuer automatique des dépôts lorsque le solde dépasse une valeur limite. Réponse □

Slide 21

Abstraction et héritage

La combinaison des modules et des classes devient incontounable lorsqu'il y a un besoin simultané d'abstraction et d'héritage :
- Les modules réalisent l'abstration par excellence.
- Les classes permettent l'héritage avec son mécanisme de liaison tardive.

Slide 22

Fonctions et classes amies

Le fait de cacher les variables d'instances dans les objets fournit un mécanisme d'abstraction de valeur. On peut ajuster cette abstraction en rendant visible en écriture ou en lecture certaines parties par des méthodes appropriées.

Toutefois, l'abstraction offerte par les objets est du tout ou rien :
- Tous les objets ont accès à la partie rendue visible.
- Aucun autre objet que self n'a accès à la partie restant cachée. Pas même les objets de la même classe.

Une solution : avoir des amis de confiance :
- rendre la représentation visible pour comminiquer entre amis
- rendre la représentation abstraite (les amis ont la même vue) pour la sécurité, en utilisation les modules.

Slide 23

Les objets amis d'une même monnaie

Slide 24

```
module type MONNAIE = sig
  type t
  class m : float -> object ('a)
    method plus : 'a -> 'a
    method prod : float -> 'a method v : t
  end
end
module Monnaie = struct
  type t = float
  class m x = object (_ : 'a)
    val v = x method v = v
    method plus(z:'a) = {< v = v +. z#v >}
    method prod x = {< v = x *. v >}
  end
end;;
```

Objets amis via l'abstraction des modules

Slide 25

La solution repose entièrement sur les modules et fonctionne comme dans la version modulaire de la monnaie

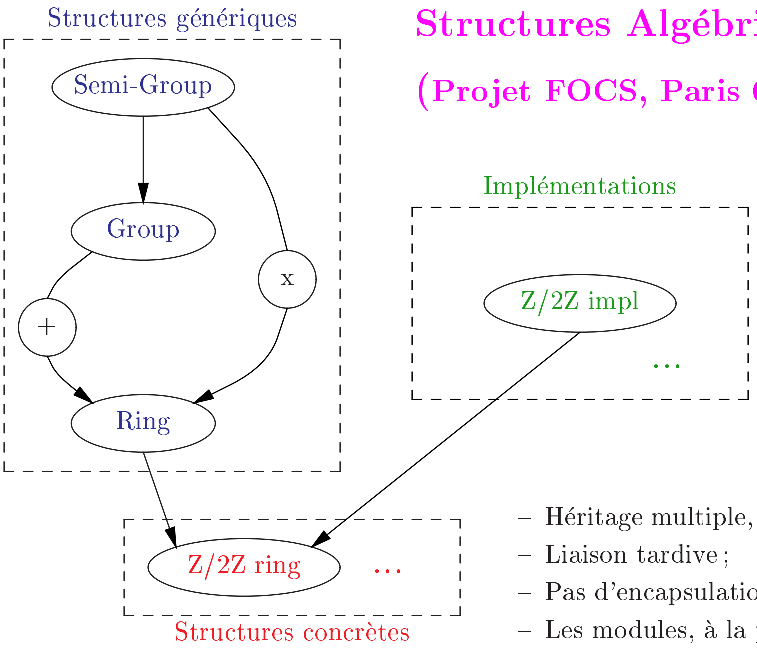
```
module Euro = (Monnaie : MONNAIE);;
let cent = new Euro.m 100.0;;
let deux_cents = cent # prod 2.0;;

module Dollar = (Monnaie : MONNAIE);;
let box = new Dollar.m 1.0;;

box # plus deux_cents;;
```

Un autre exemple :

- Les ensembles avec une opération d'union.
- Les tables de hachage avec une opération de fusion.

Un exemple de combinaison intime.

Slide 26



Structures Algébriques

(Projet FOCS, Paris 6)

– Héritage multiple,
– Liaison tardive ;
– Pas d'encapsulation de l'état ;
– Les modules, à la place.

Solutions envisagées

À base de classes
- Difficulté de l'abstraction de la représentation.
- Dissymétrie des méthodes binaires

À base de modules
- Absence d'héritage (définition incrémentale)

Slide 27

- Absence de liaison tardive (ré-implémentation plus efficace d'une implémentation par défaut).

Solution retenue Mixte
- Les classes fournissent l'héritage et la liaison tardive.
- Les modules fournissent l'abstraction (et aide à l'organisation du code)

Solution combinée

Les structures algébriques peuvent être implémentées de façon incrémentales par de petits enrichissements.

Pour pouvoir étendre et spécialiser une implémentation par défaut, on présente l'ensemble des opérations dans une classe sans variable d'instance (essentielles).

Slide 28

Chaque espèce algébrique peut être présentée dans une structure avec :

1. Les opérations par défaut fournies dans une classe virtuelle.
2. Le type des opérations manquantes (type de class)
3. Le type d'une instance de cette espèces après que toutes les opérations manquantes auront été fournies.
4. Un wrapper (sous-forme de foncteur) qui assemble les opérations par défaut et les opérations fournies et cache l'implémentation dans le résultat final.

Structures Génériques

Les structures génériques spécifient les opérations de bases et fournissent des opérations dérivées.

Slide 29

```
class virtual ['a] semi_group =
  object(self)
    method virtual equal: 'a -> 'a -> bool
    method not_equal x y = not (self#equal x y)
    method virtual zero: 'a
    method virtual plus: 'a -> 'a -> 'a
  end;;
```

Le paramètre `'a` est le type de la représentation des éléments de la structure.

Structures générique (héritage)

Les structures génériques sont construites par héritage.

Slide 30

```
class virtual ['a] group =
  object(self)
    inherit ['a] semi_group
    method virtual opposite: 'a -> 'a
    method minus x y = self#plus x (self#opposite y)
  end;;
```

Elles utilisent la liaison tardive pour définir ou redéfinir des méthodes spécifiés ou fournies avec une implémentation par défaut.

Structures concrètes

Ce sont des instances des structures génériques qui ont des implémentations concrètes pour les opérations de base.

Le groupe des entiers modulo p son implémemtation

Slide 31

```
class z_pz_impl p = object
    method equal (x : int) y = (x = y)
    method zero = 0
    method plus x y = (x + y) mod p
    method opposite x = p - 1 - x
  end;;
```

est combinée à la structure abstraite :

```
class z_pz p = object
    inherit [int] group
    inherit z_pz_impl p
  end;;
```

Abstraction de la représentation

Par les modules (on ajoute une injection une projection)

Slide 32

```
module type GROUP = sig
  type t
  val meth: t group
  val inj: int -> t val proj : t -> int
end;;
module Z_pZ (X: sig val p : int end) : GROUP =
  struct
    type t = int
    let meth = new z_pz 2
    let inj x =
      if x >= 0 && x < X.p then x else failwith "Z_pZ.inj"
    let proj x = x
  end;;
```

Motif de programmation

On peut améliorer l'exemple en définissant un motif de programmation, *i.e.* en élaborant un modèle d'assemblage des structures génériques et concrètes que reproduite pour chaque type de struture.

On peut aussi augmenter la réutilisabilité en fournissant fournir les opérations sous forme de méthodes privées, ce qui permet des les oublier ; elles sont alors rendues publiques juste avant la création d'objets représentant les structures concrètes.

Slide 33

Exercices

Ensemble Reprendre les ensembles en style objet et ajouter une méthode `fusion`.

Les polynômes

Reprendre la librairie sur les polynômes en mélangeant modules et objets.

- Définir les anneaux comme une classe (virtuelle) qui étend la classe des groupes.
- Définir les polynômes comme une classe (virtuelle) qui étend la classe des anneaux.
- Implémenter les polynômes à coefficients sur un anneau.

Utiliser les modules comme précédemment pour préserver l'abstraction.

Slide 34

4

Les points délicats de la programmation à objets.

Slide 1

Cours	Exercices
1. Les mécanismes de liaison 2. Héritage multiple et wrappers 3. Récursion, self, self-type et le clonage 4. Le Sous-typage 5. Les méthodes binaires	1. Surcharge 2. Sous-typage 3. Les tuyaux 4. Sauvegarde 5. Les vrais points

Les mécanismes de liaison.

Les différents mécanismes de liaison

Slide 2

Le mécanisme de liaison décrit l'effet d'une définition de variable et de son utilisation dans un programme.

La liaison statique donne à la variable une valeur dans une portée statique.

La liaison dynamique a une portée dynamique ; elle peut être remplacée pendant le déroulement du programme par une autre définition, changeant le sens des prochaines utilisations de cette variable.

La liaison tardive est un mécanisme propre à la programmation avec objets.

La surcharge permet d'avoir plusieurs définitions simultanées associées à une même variable (ou méthode). Le choix de la définition à considérer dépend du type (statique ou dynamique) du (ou des) arguments passés à la fonction.

Liaison statique

Slide 3

Dans Ocaml, la liaison est statique : la valeur associée à une variable est fixée définitivement à la première définition de cette variable ; elle ne peut pas être affectée par des liaisons ultérieures.

Attention ! Une liaison peut en cacher une autre ! mais ne peut pas la changer.

```
let x = 1;;
```

val x : int = 1

```
let f y = x + y;;
```

val f : int -> int = <fun>

```
f 0;;
```

− : int = 1

```
let x = 3;;
```

val x : int = 3

```
f 0;;
```

− : int = 1

La plupart des langages de programmation utilisent (heureusement) la liaison statique.

Liaison dynamique

Slide 4

La liaison dynamique signifie que la valeur d'une variable est prise à l'exécution et utilise la dernière définition de cette variable.

La liaison dynamique peut être simulée par la liaison statique en remplaçant les variables par des références vers des variables.

```
(* la première fois *)
let x = ref 1;;
```

val x : int ref = {contents=1}

```
let f y = !x + !y;;
```

val f : int ref -> int = <fun>

```
f (ref 0);;
```

− : int = 1

```
(* les fois suivantes *)
x := 3; x;;
```

− : int ref = {contents=3}

```
f (ref 0);;
```

− : int = 3

Lisp (par exemple le Lisp de Emacs) utilise la liaison dynamique. C'est une erreur de jeunesse...

Liaison dynamique (danger)

Slide 5

Toute variable est une référence : le sens d'une expression ne peut pas être compris sans le programme tout entier.

Deux évaluations successives de `f 0` produisent des résultats différents parce qu'entre temps `x` a été redéfini.

On ne peut plus garantir aucun invariant... sans avoir le programme tout entier.

La liaison statique limite les effets globaux à des objets mutables. Une fonction ne contenant pas de variables mutables possède une sémantique indépendante de son contexte d'utilisation. On dit qu'elle est *référentiellement transparente*.

La liaison dynamique n'est jamais référentiellement transparente.

Liaison tardive

Slide 6

La liaison *tardive* (ou *retardée*) est un mécanisme propre aux langages à objets, qui consiste à retarder la liaison jusqu'au moment de la création des objets. (Attention, ce mécanisme est parfois appelé *à tord*, liaison dynamique).

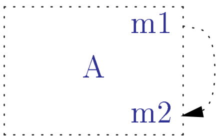
Dans une classe `A`, une méthode `m1` peut appeler une méthode `m2` de la même classe en envoyant un message `m2` à `self`. Cela simule un appel récursif, mais le *câblage* de cet appel ne sera réalisé qu'au moment de la fabrication de l'objet.

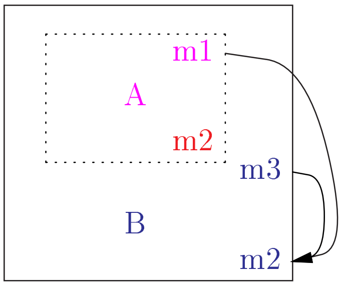
En effet, une sous-classe `B` de `A` peut redéfinir la méthode `m2`. Dans un objet de `B`, c'est la nouvelle définition de `m2` qui sera appelé par la méthode `m1`.

La liaison tardive est au cœur de la programmation avec objets. De son expressivité et de ses difficultés.

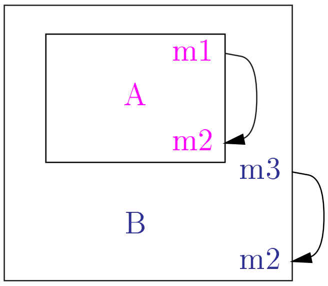
Liaison tardive

Slide 7





Liaison retardée de b. La méthode "b" est masquée (*overridden*).



Liaison non retardée : la méthode `b` est privée et cachée ou bien la classe B contient un object instance de A.

Liaison tardive (danger)

Slide 8

La liaison tardive limite l'aspect dynamique aux classes. De plus si les classes ne sont pas de première classe, une classe n'a qu'un nombre fini de classes parentes connues statiquement.

L'effet retardé de la liaison est plus contrôlable que celui de la liaison dynamique... à condition de connaître les classes parentes.

Lorsqu'on ignore l'implémentation d'une classe parente, il est impossible de connaître l'effet de la redéfinition d'une méthode sur les autres méthodes, et cela peut parfois avoir des conséquences très inattendues.

La liaison tardive est délicate, souvent difficile à contrôler, mais c'est sans doute le prix à payer pour l'expressivité. L'héritage repose sur l'optimisme (que la classe parente fera bien ce que l'on imagine à la seule vue de son interface).

Liaison tardive (variables)

Slide 9

En Ocaml, les variables d'instances sont héritées mais ne sont jamais en liaison tardive.

L'utilisation d'une variable d'instance dans une méthode réfère à la variable d'instance définie au-dessus (éventuellement dans une classe parente) et la plus proche.

La redéfinition d'une variable d'instance ne change pas la valeur de la précédente mais revient à faire une nouvelle définition qui sera utilisée par les méthodes définies en dessous (éventuellement dans une sous-classe)

```
class i = object val x = 1 method i = x end
class ii = object inherit i val x = 2 method ii = x end;;
let p = new ii in p#i, p#ii;;
```

− : int ∗ int = 1, 2

Liaison tardive (méthode privées)

En Ocaml, les méthodes visibles sont toujours en liaison tardive

Seules les méthodes privées peuvent être cachées. En effet, les méthodes publiques peuvent avoir été utilisées récursivement ; les cacher permettrait de les redéfinir avec un type incompatible.

Slide 10

Exercice 12 *Construire un tel exemple.* Réponse
Vérifier qu'il n'est pas typable en Ocaml. □

Les méthodes privées peuvent être cachées : leur liaison est alors résolu. Une redéfinition ultérieur créera une nouvelle méthode.

Les méthodes privées peuvent aussi être rendues publiques.

Liaison tardive (contrôle)

Comment arrêter (résoudre) la liaison tardive dans une classe ?

En Java

- Une méthode finale ne peut plus être redéfinie (Erreur de type).
- On peut appeler la méthode d'une classe particulière.

En Ocaml

Slide 11

- Utiliser une fonction plutôt qu'une méthode (*i.e.* une méthode statique en Java).
- Utiliser une méthode privée cachée auxiliaire.
- Remplacer l'héritage par la délégation : créer un objet de la classe parente A dans la sous-classe B.

Liaison tardive (protection préventive)

Slide 12

On se prémunie contre la redéfinition d'une méthode dans une sous-classe, en utilisant une méthode privée (gauche). On peut aussi utiliser une fonction, *i.e.* une méthode statique (droite) :

```
class type ab = object method a : int method b : int end;;
```

```
class a : ab =
  object (self)
    method a = 2 * self#b'
    method private b' = 1
    method b = self#b'
  end;;
```

```
class a =
  let b' = 1 in
  object (self)
    method a = 2 * b'
    method b = b'
  end;;
```

Liaison tardive (protection préventive)

Slide 13

Une classe héritée ne peut plus changer le comportement de `a` par inadvertance, *i.e.* en redéfinissant celui de `b`.

```
class b =
  object (self)
    inherit a
    method c = 10 + self#b
    method b = 3
  end;;
let x = new a and y = new b in x#a, y#a, y#c;;
```

− : int ∗ int ∗ int = 2, 2, 13

Liaison tardive (protection curative)

Par un relais Si la classe `a` n'est pas protégée, on peut empêcher la redéfinition de `b` de briser les invariants de la classe `a` en donnant un accès indirect aux méthodes de A par délégation, *i.e.* on fait suivre les messages vers une instance de `A`.

Slide 14

```
class a =
  object (self)
    method a = 2 * self#b
    method b = 1
  end;;
```

```
class b =
  object (self)
    val relais = new a
    method a = relais#a
    method c = 10 + self#b
    method b = 3
  end;;
```

```
let x = new a and y = new b in x#a, y#a, y#c;;
```

$- : int * int * int = 2, 2, 3$

Liaison tardive (Exemple)

Slide 15

Solution préventive

```
class type portail_blindé = object
  method clé : string -> bool
  method toc_toc : string -> string
end
class portail mot_de_passe : portail_blindé = object (self)
  method private clef (s : string) = (s = mot_de_passe)
  method toc_toc k =
   if self#clef k then "entrez!" else "sortez!"
  method clé s = self#clef s
end;;
```

La redéfinition de la méthode `clé` dans une sous classe n'affectera pas la version privée `clé_cachée`.

Liaison tardive (Exemple)

Slide 16

Solution préventive (variante)

```
class portail mot_de_passe =
  let clef (s : string) = s = mot_de_passe in
  object (self)
    method toc_toc k = if clef k then "entre!" else "dehors!"
    method clé s = clef s
  end;;
```

Structure stratifiée

```
class porte = object
  val parent = new portail "c'est moi"
  method toc_toc = parent#toc_toc
  method clé s =
    parent#clé s or parent#clé (String.uppercase s)
end;;
```

Liaison surchargée

Slide 17

La surcharge est présente dans certains langages avec ou sans objets, avec ou sans mécanisme de sous-typage. Il n'y a pas de surcharge en Ocaml, il y en a en Java.

La liaison surchargée est statique mais une variable peut avoir plusieurs définitions simultanées avec des types différents.

La surcharge est souvent réservée aux fonctions, et seulement sur les types des arguments. La résolution est alors effectuée en fonction du type des arguments.

Liaison surchargée (résolution)

La résolution de la surcharge, consiste à choisir quelle définition de la variable utiliser. Ce choix est fait à chaque utilisation de la variable et dépend de son contexte.

Résolution statique

Slide 18

La résolution statique utilise les types connus à la compilation. Les types des valeurs n'ont pas besoin d'être passés à l'exécution.

Résolution dynamique

La résolution dynamique utilise les types à l'exécution, ce qui oblige à les conserver.

En Java, une partie de la surcharge est résolue dynamiquement, mais uniquement avec l'information de type statique.

Liaison surchargée (exemple en Java)

Slide 19

```
class Point {
  int x = 0;
  Point (int x0) { x = x0; }
}
class Bipoint extends Point {
  int y = 0;
  Bipoint (int x0) { super (x0); }
  Bipoint (int x0, int y0) { super (x0); y = y0; }
}
```

On peut créer des bipoints en passant un ou deux arguments et le constructeur de classe correspondant sera appelé :

```
Bipoint p = Bipoint(1); Bipoint q = Bipoint(1,2)
```

Dans le premier cas, le point aura pour abscisse sa valeur par défaut `0`.

Liaison surchargée (avantage)

Slide 20

Surcharge statique

La surcharge est le plus souvent statique.

Dans ce cas, elle n'augmente pas l'expressivité, mais la convivialité. Elle permet de donner le même nom à plusieurs variantes d'une même fonction.

Surcharge dynamique

Elle augmente l'expressivité du langage.

Par exemple, une autre approche de la programmation avec objets traite les méthodes comme des fonctions surchargées. L'envoi d'un message est alors résolu dynamiquement en regardant le type des objets reçus en argument à chaque appel de méthode.

Liaison surchargée (danger)

Slide 21

Sémantique typée

La sémantique du programme est déterminée par le typage. Si la surcharge est statique, le typage n'est qu'une approximation de la valeur. Au cours du calcul le typage devient plus précis.

Par exemple, il est fréquent de remplacer un programme par le programme obtenu après une étape d'évaluation, *i.e.* faire de l'évaluation partielle, manuellement ou automatiquement.

Dans le cas de la surcharge résolu statiquement, cette transformation n'est plus valable sans précautions : maintenir artificiellement une information de typage approchée (ce qui n'est pas toujours possible).

Liaison surchargée (dangers)

Évaluation partielle

Slide 22

L'évaluation partielle consiste à remplacé un appel de fonction connu statiquement par le résultat de cet appel.

```
class Programme {
  static boolean surcharge (Object x) { return true; }
  static int surcharge (String s) { return 42; }
  static void eval (Object x)
    { System.out.println(surcharge(x)); }
  static void call () { eval("hello"); }
}
```

Ici `hello` est passé à la `eval`, donc coercé en un `Object` et passé à `surcharge` avec le type statique `Objet`. Le compilateur choisi donc la première définition et `Programme.call` retourne `true`

Liaison surchargée (danger)

Version spécialisée incorrecte

Slide 23

```
class Programme_partiellement_evalue {
  ...
  static void call () {
     System.out.println(surcharge("hello"));
  }
}
```

Ici `hello` est directement passé à `surcharge` avec le type `String`. Le compilateur choisit donc la deuxième définition.

Pourtant dans l'intuition du programmeur en écrivant eval est de propager la surcharge.

Surcharge (danger)

Version spécialisée correcte

Slide 24

```
class Programme_partiellement_evalue {
  ...
  static void call () {
    Object x = "hello";
    System.out.println(surcharge(x));
  }
}
```

Ici `hello` est directement passé à `surcharge` avec le type `Object`, comme dans la version de référence. Le compilateur choisit donc la première définition.

Liaison surchargée (exercice)

Slide 25

```
class A          { int bin(A x){ return 1; } }
class B extends A { int bin(B x){ return 2; } }

A aa = new A(); B bb = new B(); A ab = bb;
```

Exercice 13 *Quelle est la valeur retournée par* `x.bin(y)` *pour chacune des 9 combinaisons possibles de "(x,y)" ?* Réponse

Écrire un programme `Java` *qui permet de vérifier les cas ci-dessus (en calculant les 9 combinaisons).* Réponse

On considére maintenant la définition suivante :

```
class A          { int bin(A x){ return 1; } }
class B extends A {
  int bin(A x) { return 3; }
  int bin(B x) { return 2; }
}
```

Quelle est la différence "d'effet" entre les deux méthodes définies dans la classe `B`*.* Réponse

Répondre à la première question mais avec la définition ci-dessus. Réponse

Écrire une variante qui retourne toujours (et seulement) `2` *lorsque les deux objets sont de la classe* `B` *(et seulement* `1` *autrement)* Réponse

La surcharge peut être éliminée statiquement en choisissant des nom non-ambigü (par exemple en suffixant les nom par le type des arguments) : écrire une version du programme de la question 2 qui n'utile pas la surcharge. Réponse

Slide 26 *Quelle version de* `bin_A` *ou* `bin_B` *faut-il utiliser pour les différents appels (9 combinaisons possibles) pour être le plus précis possible ? (On indiquera par une prime si la version exécutée est celle définie dans la class A ou dans la class B).* Réponse □

Héritage multiples, et wrappers (class mixins).

Héritage simple *v.s.* multiple

Avec l'héritage simple, une classe a au plus un parent. La hiérarchie des classes forme donc un arbre.

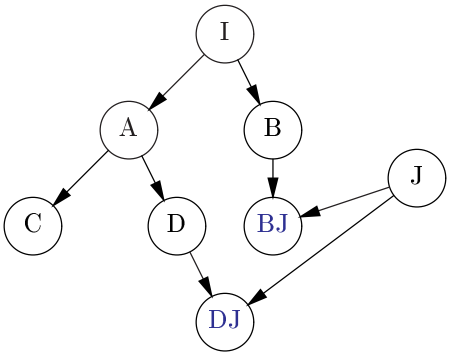
Avec l'héritage multiple, une classe peut avoir plusieurs parents. La hiérarchie des classes forme un graphe acyclique.

Slide 27 C'est une généralisation naturelle de l'héritage simple, mais elle rend la compilation plus difficile.

Comparaison

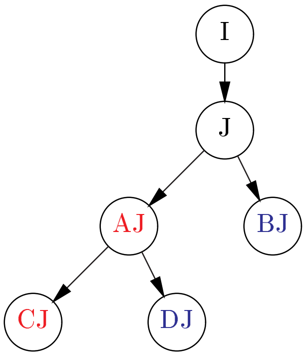
Slide 28

Héritage multiple



Extension des classes B et D par J, a posteriori, sans modification des autres classes.

Héritage simple



Les classes A,B,C et D sont pollués par la classe J. Il faut changer le code des classes A et B.

Quelques exemples

La sauvegarde

Construction du groupe des entiers

Constructeurs de classes (wrappers)

Les pipes

Slide 29

Les pipes

Écriture

```
class virtual writer = object (self)
  method virtual put_char : char -> unit
  method put_string s =
    for i = 0 to String.length s - 1
    do self#put_char s.[i] done
end;;
class fileout filename = object
  inherit writer
  val chan = open_out filename
  method put_char c = output_char chan c
end;;
```

Slide 30

Héritage multiple (pipes)

Slide 31

Lecture

```
class virtual reader = object (self)
  method virtual get_char : char
  method get_string n =
    let s = String.create n in
    for i = 0 to n - 1 do s.[i] <- self#get_char done;
    s
end;;
class filein filename = object
  inherit reader
  val chan = open_in filename
  method get_char = input_char chan
end;;
```

Héritage multiple (pipes)

Slide 32

Pipe

```
class pipe = object
  val q = Queue.create ()
  inherit writer()
  inherit reader()
  method put_char c = Queue.add c q
  method get_char =
    try Queue.take q with
      Queue.Empty -> raise End_of_file
end
```

Exercice 14 *Reprendre l'exemple des* pipes *en le rendant plus réaliste, par exemple, ajouter une fonction d'écriture et de lecture des entiers.* □

Les wrappers (class mixins)

Slide 33

Les *(class) mixins* ou *wrappers* désigne la possibilité d'abstraire des classes par rapport à d'autres classes.

```
class w (arg : arg_type) = body end
class b = w c
```

Ce n'est pas directement possible en Ocaml au niveau du langage de classe. Toutefois, on peut obtenir l'effet des mixins en utilisant des modules (expressif mais un peu lourd) ou l'héritage multiple (léger, mais pas exactement les différences).

Utilisation des modules

```
module W (Arg : sig class class_arg : class_arg_type end) =
  struct open Arg class resultat = body end
class b = Wrapper(struct class arg = c end).resultat
```

`b` ne voit que les méthodes de `arg_type` et de `body` (pas de `arg`).

Exemple

Slide 34

```
module Secure
    (Arg : sig class c : object
      method retrait : float -> float
      method dépôt : float -> unit
      method solde : float
    end end) =
  struct
    let interdits_bancaire = ref []
    let gelé x = List.mem x !interdits_bancaire
    class c = object (self)
      inherit Arg.c as super
      method retrait x =
        if gelé (self :> Arg.c) then raise (Failure "interdit bancaire")
        else super#retrait x
    end
  end;;
module Secure_compte = Secure (struct class c = compte end);;
class secure_compte = Secure_compte.c;;
```

Wrappers (simulation de l'héritage multiple)

Slide 35

Les wrappers permettent de simulé l'héritage multiple (dans les langages qui n'en ont pas) :

```
class c1 = body1
class c2 = inherit c0 body2
class c3 = object inherit c1 inherit c2 body3 end
```

est implémenté par :

```
class c1 (s) = object inherit s body1 end
class c2 (s) = object inherit c0(s) body2 end
class c3 (s) = object inherit c2(c1(s)) body3 end
```

et en remplaçant `new c1` par `new c1(object end)`.

Si le type de `s` *doit être fixé, ce qui est en général le cas, cette solution est restreinte et ne remplace pas l'héritage multiple.*

Wrapper *v.s.* héritage multiple

Slide 36

Les wrappers

– Il faut prévoir le besoin d'abstraction.
– Il faut connaître l'interface de l'argument (les composantes non spécifiées qui peuvent être oubliées seront cachées).
+ Avantage : le câblage (liaison de super, overriding) peut-être fait dans le wrapper et être partagé.
+ Les variables de la classe parente sont visibles.

L'héritage multiple

+ Rien à prévoir
+ Il y a toujours une classe parente hypothétique (héritage)
– Le câblage doit être réalisé après coup à chaque utilisation
– Les variables ne peuvent pas être virtuelles

Les deux sont complémentaires, avec un recouvrement important.

Récursion, self, son type et le clonage..

Self (lui-même)

Slide 37

Self désigne pendant l'exécution d'une méthode l'objet qui a appelé cette méthode. En Java cet objet est désigner par le mot clé `this`. En Ocaml, on doit déclarer en tête de la classe une variable pour désigner cet objet. Nous parlerons de `self` ici quelque soit le mécanisme de liaison.

"self" peut donc représenter un objet de la classe mais aussi d'une sous classe.

Envoyer un message à `self` c'est envoyer un message à l'objet en train d'exécuter une méthode, ce qui réalise la récursion. Parce que cette méthode est prise dans l'objet à l'exécution (du moins en théorie, le compilateur étant libre de compiler l'appel autrement pourvu que son effet soit indiscernable) et non dans la classe, un message à self produit une liaison tardive.

Le type de self

Slide 38

Dans une classe, le type de self, est celui d'un objet ayant toutes les méthodes de la classe et peut-être d'autres (par exemple celles ajoutées dans une sous-classe).

```
class c = object (self) method m = self end;;
```

class c : object ('a) method m : 'a end

En Ocaml, le type de self est polymorphe ; il contient une variable de rangée. On peut voir le type de self d'une classe `c` en faisant :

```
fun x -> (x :> c);;
```

− : (< m : 'a; .. > as 'a) -> c = <fun>

Ici, c'est un type récursif `'a` d'un objet qui a une méthode `m` qui retourne un objet de type `'a`.

Le type de self (suite)

Slide 39

En Ocaml, le type polymorphe de self lui permet dans une sous-classe d'être une instance de celui de la classe parente.

```
class cd = object (self) inherit c method n = 2 end;;
```

class cd : object ('a) method m : 'a method n : int end

```
fun x -> (x :> cd);;
```

− : (< m : 'a; n : int; .. > as 'a) -> cd = <fun>

Aussi self dans un objet de la sous-classe aura bien le type de la sous-classe et non celui de la classe parente.

```
new m1 # m1;;
```

− : m1 = <obj>

```
new m1_m2 # m1;;
```

− : m1_m2 = <obj>

Le type de self en Java

En java, il n'y pas de type pour self (this) et on lui donne le type de la classe courante : en particulier, une méthode qui retourne `this` garde dans une classe héritée le type de la classe parente.

Slide 40

```
class C {
   C () { }
   C m() { return this; }
}
```

```
class CD extends C {
   CD () { }
   int n() { return 2; }
}
```

```
class Error {
   static CD cd = new CD();
   static int x = (cd.m()).n();
}
```

Types récursifs

Les types des objets sont récursifs, car une méthode doit pouvoir retourner un objet du même type.

En Ocaml un type récursif inféré est forcément un type objet et décrit par une équation `(< m1 : int; m2 : 'a > as 'a)` avec le mot clé `as`. Ici, la variable `'a` sert uniquement à décrire la récursion : elle n'est pas polymorphe car il n'y a pas d'autre variable dans le membre gauche.

Slide 41

Par contre, le type ouvert `(< m1 : int; m2 ; 'a; ..> as 'a)` est polymorphe, car `..` représente une variable de type anonyme.

En Ocaml, il existe aussi des types récursifs déclarés :

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

Types récursifs

En java les types peuvent aussi être récursifs[a] mais il sont toujours déclarés :

Slide 42

```
class Point {
  int x;
  Point (int x0) { x = x0; }
  Point self () { return this; }
}
```

Cette classe définit un type d'objet `Point` ayant un champ `x` de type `int` et une méthode `self` de type `Point`.

[a]Tout langage de programmation intéressant a des structures de données récursives, donc des types récursifs pour les décrire.

Le type de self (avantages)

Slide 43

Le type de self permet de traiter correctement
- les méthodes qui retournent self.
- les méthodes binaires.

Overriding La construction `{< l_1 = e_1; ... l_p = e_p >}` permet de retourner une copie de self dans lesquels les variables `l_i` sont liée aux valeurs résultant de l'évaluation de `e_i`.

Application : il permet de créer une nouvel objet du même type que self.

En effet les constructeurs de la classe retournent un objet de la classe courante et ils feront de même dans une sous-classe. (La construction `Oo.copy` ne permet pas de changer les variables d'instances.)

Clonage

Un résumé des opérations de clonage et de leurs effets

Slide 44

```
class démon =
  let population = ref 0 in
  object (lui_même)
    val mutable gènes = Random.int 9999999 val au_delà = ref 0
    method pensée x = au_delà := x
    method intuition = !au_delà
    method identité = gènes
    method self = lui_même
    method clone = lui_même#contrôle; {< >}
    method reproduction = lui_même#contrôle; {< gènes = gènes + 1 >}
    method mutation = gènes <- gènes + Random.int 9999999
    method private contrôle =
      if !population < 10000 then incr population
      else raise (Failure "surpopulation")
    initializer lui_même#contrôle
  end;;
```

Clonage

Slide 45

La variable de classe `population` est identique pour tous les objets de la classe.

La variable `au_delà` est identique pour tous les objets issus d'un même object (par clonage ou reproduction). Elle permet de communiquer des informations entre de tels objets. Par exemple, des objets graphiques produits par clonage partageront des variables communes (couleur, tailler, etc.) et la modification de la propriété sur l'un de ces objets affectera tous les autres objets de la même famille.

La variable `gènes` est propre à chaque objet (situation la plus fréquente).

Exercice 15 *(Voir l'exercice sur la sauvegarde)* □

Le Sous-typage.

Qu'est-ce que c'est ?

Le sous-typage est un affaiblissement de l'information statique de type qui sous certaines conditions permet de voir des objets de types différents sous un même type.

La relation de sous-typage est toujours fermée par reflexivité et transitivité.

Slide 46

Sous-typage structurel Les types sont construits librement à partir de types de base, et la relation de sous-typage est définie une fois pour toute et ne dépend que de la structure des types.

Sous-typage déclaré Les types sont vus comme des atomes et la relation de sous-typage est construite au fur et à mesure de la définition de nouveaux types par des déclarations manuelles où automatiques.

Bien sûr, le typeur vérifie que les déclaration de sous-typage sont permines.

On peut combiner sous-typage structurel et déclaré.

(Mais c'est autant un cumul des difficultés que des avantages.)

Abbréviations de types

Les abbréviations de types sont transparentes : le sous-typage entre types abbrégés est donc structurel ou par nom mais indépendamment du mécanisme d'abbréviation.

Slide 47

Intuition

Considérons, par exemple, une relation de sous-typage structurelle entre types-enregistrements.

Sous-typage en largeur

Un enregistrement avec plus de champs peut toujours être utilisé à la place d'un enregistrement avec moins de champs. Au niveau des types, on peut donc cacher sans risque d'erreurs un nombre arbitraire de champs.

Slide 48

La relation de sous-typage correspond à l'inclusion des champs.

Sous-typage en profondeur

Si un élément de type A peut être utilisé comme un élément de type B, alors une liste d'éléments de type A peut aussi être utilisée comme une liste d'éléments de types B.

Contra-variance

Co-variance On dit que l'opérateur des listes est contra-variant, parce qu'il propage la relation de sous-typage entre les éléments en une relation entre des listes de ces éléments dans le même sens.

Slide 49

Contra-variance Cette propagation est inversée dans le cas des consommateurs : Si un élément de type `A` peut être utilisé comme un élément de type `B`, alors une fonctions dont l'argument est de type `B` peut être coercée en une fonction dont l'argument est de type `A`.

En effet, un argument de type `A` peut être vu par sous-typage comme un argument de type `B`, donc passé à la fonction.

Le type des fonctions

Il est co-variant à droite et contra-variant à gauche.

Non-variance

Slide 50

La propagation de la relation au travers d'un constructeur n'est pas possible dans certains cas. Ces types n'ont qu'eux-mêmes pour sous-types. Le constructeur de type est dit non-variant.

Par exemple, imaginons qu'on définisse le type suivant :

```
type 'a transformation = ('a -> 'a)
```

Le constructeur de type `transformation` est non-variant, car pour qu'une `A transformation` soit plus petit qu'une `B transformation`, il faudrait que `A -> A` soit plus petit que `B -> B`, *i.e.* que `A` soit à la fois plus grand et plus petit que `B`. Ce qui n'est vrai que pour `A` égal à `B` (ici, on suppose que la relation de sous-typage est anti-symmétrique).

Les types mutables

Pour des raisons similaires, le type `'a buffer` défini par

```
type 'a buffer = ('a -> unit) * (unit -> 'a)
```

est non-variant.

Une référence est un tampon à un élément. Il se comporte comme un objet avec deux méthodes de types respectifs `'a -> unit` et `unit -> 'a`. Ainsi le constructeur de référence `'a ref` est non-variant.

Slide 51

Plus généralement, une définition de type est co-variante (resp. contra-variante) en une variable si toutes les occurrences de cette variable sont co-variantes (resp. contra-variantes)

Par défaut, *i.e.* en l'absence d'information, un constructeur de type doit être considéré comme non-variant.

Sous-typage en Ocaml

Le sous-typage est structurel. Il est définit formellement comme la plus petite relation réflexive et transitive fermée par les opérations suivantes :
- Si $A' < A$ et $B < B'$ alors $A \to B < A' \to B'$,
- Si $A_i < B_i$ alors $\langle \ell_1 : A_1; ...\ell_k : A_k; ...l_n : A_n\rangle < \langle \ell_1 : B_1; ...\ell_k : B_k\rangle$

On retrouve la contra-variance à gauche de la flèche et la co-variance à droite de la flèche, plus le sous-typage en largeur pour les types objets.

Slide 52

Comme les types objets ne montrent pas les types des variables, il n'y a pas de cas particulier à faire pour les champs mutables.

Si un champ peut être lu et écrit de l'extérieur, cela revient à donner deux fonctions de lecture et d'écriture de variances opposées, ce qui rend le type de l'objet non-variant par rapport au type du champ mutable.

Exercice 16 *Définir une classe paramétrique `['a] cell` des cellules (version object des références) de type `'a`.* Réponse
Expliquer pourquoi le type `'a cell` est non-variant par rapport à `'a`. Réponse

Vérifier expérimentalement que c'est bien le cas. Réponse

Que faut-il changer dans la classe `['a] cell` *(en perdant certaines fonctionnalités) pour que la classe devienne co-variante ? contra-variante ?* Réponse

Le type des objets d'une classe héritée de `['a] ref` *peut-il être sous-type de* `[a'] ref` *?* Réponse

Slide 53

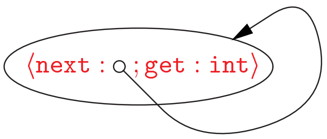
```
class ['a] cell' x =
  object (self)
    inherit ['a] cell x
    method consulte = self#set (self#get)
  end;;
fun x -> (x : 'a cell' :> 'a cell);;
```

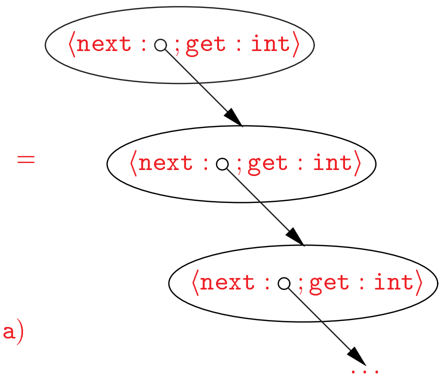
□

Types récursifs

Les types récursifs sont des arbres infinis réguliers. Un arbre régulier est un arbre qui a un nombre fini de sous-arbres. Il est représentable de façon finie en indiquant le partage de certains sous-arbres par un nommage.

Slide 54





En Ocaml, on note :
`(⟨next : 'a ; get : int⟩ as 'a)`

Types-récursifs (déroulement)

La notation avec partage cache l'arbre déroulé. Le déroulement est toujours possible (c'est la bonne façon de comprendre).

En particulier, $(\alpha \to \beta)$ `as` α est égal à $((\alpha \to \beta)$ `as` $\alpha) \to \beta$ mais aussi à $((\alpha \to \beta) \to \beta)$ `as` α qui font tous deux apparaître une occurrence négative de β.

Slide 55

Lorsque la variable muette α utilisée pour dénoter la récursion apparaît en position négative, alors toutes les occurrences de l'arbre désigné par α se répètent positivement et négativement.

Si α apparaît négativement dans t, alors t n'admet pas d'autres sous-types que lui-même.

Par exemple, un objet de la forme $\langle \ell_0 : \alpha \to \tau_0; \ell_1 : \tau_1; ...\ell_n : \tau_n \rangle$ `as` α ne sera jamais sous-type de $\langle \ell_0 : \alpha \to \tau_0; \ell_i : \tau_i \rangle$. Il faut d'abord cacher toutes les méthodes binaires.

Perte d'information

Le sous-typage est une perte d'information de type irréversible (*i.e.* non recouvrable statiquement).

C'est bien : cela permet de voir des objets de différentes classes avec une interface commune, et donc de les mettre ensemble dans une même boîte.

Slide 56

C'est embêtant : lorsque l'on retire des objets d'un telle boîte, on ne voit plus que leur interface commune, on ne peut donc plus obtenir de comportement spécifique.

Perte d'information (limitation)

Accepter la perte d'information

On n'a pas besoin de recouvrer les vrais types des objets.

Pour recouvrer l'information perdue

Utilisation de types concrets (variantes).

Slide 57

Type-case : on peut tester l'appartenance à une classe ou le respect d'une interface.

Pour éviter de perdre de l'information

Utilisation du polymorphisme (à la place ou en combinaison avec le sous-typage).

Exemple en Ocaml

Imaginons deux classes `a` et `b` dont ont veux mettre les objets dans une liste avec l'interface `c`.

Avec perte d'information :

```
let (<<) t h = (h :> c) :: t;;
let tous_ensemble = [] << a1 << b1 << a2;;
let send_à_tous x = x#mc in
List.map send_à_tous tous_ensemble;;
```

Slide 58

Avec types concrets :

```
type 'a as_c = A of a | B of b;;
let (<<) t h = h :: t;;
let chacun_pour_soi = [] << A a1 << B b1 << A a2;;
let send_à_chacun =
  function A x -> x#ma | B y -> y#mb in
List.map send_à_chacun chacun_pour_soi;;
```

Solution mixte

Slide 59

Un usage mixte, à la fois homogène et hétérogène est aussi possible :

```
type 'a as_c = A of a | B of b;;
let cA x = (x :> c), A x and cB x = (x :> c), B x;;
let tous_pour_un = [] << cA a1 << cB b1 << cA a2;;
let send_à_tous (x,_) = x#mc in
List.map send_à_tous tous_pour_un;;
let send_à_chacun =
  function _, A x -> x#ma | _, B y -> y#mb in
List.map send_à_chacun tous_pour_un;;
```

Perte d'information (limitation)

Slide 60

En Ocaml, une fonction qui prend en argument un objet avec une méthode `m` appelle cette méthode et retourne son argument :

```
let f x = x#m; x;;
```

```
val f : (< m : 'b; .. > as 'a) -> 'a = <fun>
```

Cette fonction n'utilise pas le sous-typage, mais le polymorphisme : *Pour tout objet possédant une méthode `m`, elle retourne un objet du même type de son argument*

Avec du sous-typage, sans polymorphisme, on pourrait prendre `< m : 'b >` pour le type de l'argument, mais dans ce cas, on ne peut retourner que le type `< m : 'b>`.

```
f : < m : 'b > -> < m : 'b >
```

Une solution pour retrouver l'information est le polymorphisme contraint (plus difficile) : `(f : All ('a <: <m : 'b >) 'a -> 'a)`

Utilisation de type case

Cela revient à un mécanisme de typage dynamique. Les objets doivent alors porter leur types à l'exécution, ce qui peut être coûteux.

En java, on utilise la primitive `instanceof` combinée avec `cast`.

```
if (c instanceof A) ... (A) c ...
else ... c ...
```

Slide 61

Cette combinaison est sûre (ne produit pas d'exception). Les autres usages du cast sont plus dangereux (risque d'exception).

En Ocaml, les valeurs ne portent pas leur type à l'exécution et il n'y a pas de type-case.

Le sous-typage n'est pas de l'héritage

Sous-typage n'implique pas héritage

En Ocaml, les types sont structurels, donc le sous-typage ne dépend pas de la façon dont la classe est construite : prendre par exemple deux copies de la même classe produisant deux classes différentes avec la même interface.

Slide 62

En java, une clause `implement` crée une relation de sous-typage sans relation d'héritage.

Héritage n'implique pas sous-typage

En Ocaml, une classe qui hérite d'une autre ayant une méthode binaire ne crée pas de relation sous-typage entre leur type.

Les méthodes binaires.

Qu'est-ce qu'une méthode binaire ?

Slide 63

Une méthode binaire est une méthode qui combine self avec un argument du même type que self. Typiquement :

```
class point x0 =
  object (self)
    val x = abs x0
    method getx = x
    method inf p =
      p#getx < x
    method max p =
      if self#inf p then p
      else self
  end;;
```

```
class point : int ->
  object ('a)
    val x : int
    method getx : int
    method inf : 'a -> bool
    method max : 'a -> 'a
  end
```

Une méthode binaire a comme argument un objet du même type que `self`. Par extension, une méthode est binaire si son type contient une occurrence contra-variante du type de self.

Autres exemples

L'union sur les ensembles.

La concaténation des chaînes de caractères

Append sur les listes.

Plus généralement, toutes les opérations binaires sur les types données implémentés dans un style objet...

Slide 64

Quelle particularité ?

Difficulté Les méthodes binaires posent plusieurs difficultés :
- Il est difficile de leur préserver un type correct dans une sous-classe.
- Le type des objets avec une méthode binaire n'admet pas de sous-type tant qu'un méthode binaire reste visible.

Solutions La plus répandue consiste à ignorer le problème, et donc à ne pas hériter correctement des méthodes binaires (Java).

Slide 65

Pour résoudre ce problème, il faut utiliser un système de type sophistiqué (Ocaml).

Une autre solution consiste à sortir la méthode binaire de la classe.

Méthodes binaires en Java

Slide 66

En java, une méthode binaire est typée en donnant à self le type de la classe dans laquelle elle est définie.

En conséquence, elle n'est pas héritée comme une méthode binaire : elle n'accepte plus en argument qu'un objet d'une classe parente.

Pour retrouver le bon type dans une sous-classe, il faut la redéfinir. Ou alors, il faudra utiliser des conversions de type, avec risque d'échec à l'exécution.

Méthodes binaires en Ocaml

Slide 67

Les méthodes binaires sont correctement typées grâce à l'utilisation du polymorphisme des variables de rangée.

```
class point_coloré x0 y0 =
  object (self)
    inherit point x0 as super
    val y = abs y0
    method gety = y
    method inf p =
      x + y <
        p#getx + p#gety
  end;;
```

```
class point_coloré :
  int -> int ->
  object ('a)
    val x : int
    val y : int
    method getx : int
    method gety : int
    method inf : 'a -> bool
    method max : 'a -> 'a
  end
```

Ingrédients nécessaires

Il faut typer la classe parente en supposant que self est le type d'un objet d'une sous-classe arbitraire, et sans perdre d'information sur le type de self.

Pour cela, il faut utiliser du polymorphisme, que ce soit avec variable de rangée ou avec des contraintes de sous-typage (voir la fin de la partie précédente).

Slide 68

Indépendemment du bon typage de self, le problème le sous-typage des objets avec des méthodes binaires reste un problème.

Sous-typage des méthodes binaires

En supposant, `p = new point 0` et `q = new point_coloré 0 0` et en oubliant temporairement le typage (on ne pourra donc pas les tester en Ocaml) quelles sont les combinaisons de `x.max y` qui s'évaluerait quand même correctement ?

Réponse

Slide 69

La méthode `max` de la classe `point_coloré` lit indirectement, par un appel à la méthode `inf` variable `y` de son argument donc son argument doit être de la classe `point_coloré`.

x \ y	p	q
p	OK	OK
q	×	OK

Il est donc incorrect de dire que `q` peut être vu avec l'interface d'un point coloré, sinon, on pourrait appeler sa méthode max avec `p`.

Difficulté

Slide 70

Le problème ci-dessus n'est pas facilement détectable, car l'appel à la méthode `gety` est indirect.

Le problème vient du fait que le type de self représente à la fois
(1) les méthodes implémentées,
(2) les méthodes utilisées récursivement.
(3) les méthodes utilisées dans un objet du même type que self.
Ce sont les méthodes (3) qu'il est dangereux de cacher en présence de méthodes binaires. Par exemple, `getx` ou `gety`.

En effet, les méthodes
(1) peuvent être oubliées : elles ne seront simplement pas disponibles. (par exemple `max`)
(2) sont prises dans l'objet et non dans l'argument, et donc toujours présentes. (par exemple `inf`)

Externaliser les méthodes binaires

Slide 71

On peut externaliser les méthodes binaires.

```
class petit_point x0 =
  object (self)
    val x = abs x0
    method getx = x
  end;;
let inf p q = p#getx < q#getx;;
```

```
val inf : < getx : 'a; .. > -> < getx : 'a; .. > -> bool = <fun>
```

```
let max p q = if inf p q then p else q;;
```

```
val max : (< getx : 'b; .. > as 'a) -> 'a -> 'a = <fun>
```

C'est plus une dérobade qu'une solution.

Méthodes binaires et objets amis

Typiquement, une méthode binaire a besoin de déstructurer son argument pour l'utiliser. L'argument doit donc montrer une partie de sa représentation. Ainsi, une classe définissant une méthode binaire doit typiquement révéler la représentation de ses objets.

(Penser par exemple à l'opération d'union sur les ensembles.)

Slide 72

Si l'on veut par ailleurs cacher cette représentation entre tous les objets de la même classe, on va rendre tous les objets de la même classe "amis" en un utilisant un module (voir objets amis)

Exemple

Slide 73

```
module type Point = sig
  type t
  class c : int -> int ->
    object ('a)
      method getx : t
      method inf : 'a -> bool
      method max : 'a -> 'a
    end
end;;
```

```
module Point : Point =
  struct
    type t = int
    class c x = point x
  end;;
```

Conclusion

La programmation avec objets est un paradigme idéal pour les données munies d'opérations unaires ou externes.

Elle devient plus délicate et moins bien adaptée dès qu'il y a besoin d'utiliser des méthodes binaires.

Slide 74

En générale, celles-ci détruisent la symétrie, par exemple `plus x y` devient `x#plus y`.

D'autres approches des objets qui traitent les méthodes comme des fonctions surchargées définies en dehors des objets sont mieux appropriées ici, mais ces approchent présentent d'autres difficultés.

Les vrais points

Slide 75

Mettre en place la librairie graphique

En utilisant ce `Makefile` :

```
make ocamlgraph
```

Récupérer le fichier `bonjour.ml` pour tester.

```
make testgraph
```

Pour les commandes graphiques, voir le manuel.

Ecrire quelques composants graphiques

Relever quelques composants graphiques sous forme objet en partageant leur code autant que possible : Point, Segments, Triangles, Parallélépipèdes, Cercles, Creux Pleins,

Écrire des composants d'assemblage

Assemblage horizontal, assemblage vertical. (Pour cela, il faut associer à chaque composant des points d'ancrage.)

Rendre les objets graphiques réactifs

Pour qu'il reçoivent les événements de la souris dans la zone graphique qu'il couvrent.

Slide 76

5

Programmation objet en Java.

Slide 1

Cours	Exercices
1. Classes, Constructeurs 2. Héritage 3. Surcharge 4. Les interfaces 5. Le sous-typage 6. Les conversions 7. Les packages	1. Surcharge 2. Méthodes binaires 3. Abstraction de type

Avertissement

Ce n'est pas un cours sur Java, ni sur la programmation Objet. On suppose l'essentiel du langage Java déjà connu.

On rappel simplement les choix de Java concernant la programmation objet. La comparaison avec Ocaml est souvent implicite.

Pour une spécification complète se reporter à la définition du langage :

Slide 2

> *The Java Language Specification.*
> James Gosling, Bill Joy, and Guy Steele.

disponible aussi en HTML

Les classes

```
class Point {
  int x; int y = 0;
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
}
```

Une classe est nommée (`Point`) ; son corps est une liste de déclarations en nombre et en ordre quelconques :

Slide 3

- de variables (`x`), qui peuvent être introduites avec une valeur par défaut.
- de constructeurs (`Point`), dont le corps est de type `void` ; un constructeur est reconnu au fait que son nom est le même que celui de sa classe et que son type de retour n'est pas indiqué.
- de méthodes (`bouge`) dont le corps est arbitraire.

Par exemple, la classe `class vide {}` est une classe vide sans méthodes et sans constructeurs.

Les constructeurs

On crée une instance d'une classe avec la construction `new` suivie d'un constructeur de classe et de ses arguments, par exemple `new Point (3)`.

Le typage ne garantit pas que toutes les variables sont initialisées. Cependant, le compilateur affecte aux variables non initialisées une valeur par défaut du type déclaré.

Slide 4

```
class Point {
  int x; int y = 0;
  Point () { }
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
}
```

est une définition correcte et `(new Point()).x` retourne 0.

This

Dans le corps d'une méthode, `this` désigne l'objet en train d'exécuter cette méthode.

Slide 5

```
class Point {
  int x = 0;
  Point () { }
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
  void bouge_bouge () { this.bouge(); this.bouge(); }
}
```

L'héritage

```
class Bipoint extends Point { int y;
  Bipoint () { }
  Bipoint (int x0, int y0) { super(x0); y = y0; }
  void bouge () { super.bouge (); y = y + 1; }
}
```

Slide 6

Dans une sous-classe, `super` désigne la classe parente. Un constructeur d'une sous-classe doit faire appel à un constructeur de la classe parente. Cet appel doit être la première instruction. Si ce n'est pas fait, le compilateur ajoute `super()`. Ainsi,

```
  Bipoint (int y0) { y = y0; }
  Bipoint (int y0) { super(); y = y0; }
```

sont deux phrases équivalentes (et c'eut été une erreur si `Point() { ... }` n'était pas défini dans la classe `Point`.)

La surcharge

Slide 7

Les constructeurs de classe et les méthodes sont surchargés.

Les constructeurs de classes ne sont pas hérités.

Les méthodes sont héritées. Ainsi, l'ensemble des définitions d'une méthode `m` dans une classe `A` est l'ensemble des définitions de `m` dans la classe `A` et dans la classe parente.

Lorsqu'une sous-classe redéfinit une méthode présente dans la classe parente avec le même type, la nouvelle définition cache (*override*) la précédente. Sinon elle ne fait que la surcharger.

Pour la résolution de la surcharge, voir le cours sur la liaison surchargée.

Interfaces et implémentations

Les interfaces sont des spécifications des implémentations.

```
interface Printable {
  void print ();
}
```

L'interface `Printable` n'est pas une classe. Elle n'a pas d'implémentation propre. Mais l'interface peut être implémentée (*i.e.* respectée) par différentes classes.

Slide 8

Une classe peut implémenter plusieurs interfaces.

```
class Printable_point extends Point implements Printable {
  Printable_point () { super(); }
  public void print () { System.out.println(x); }
}
```

Utilisation

La déclaration d'une interface I dans la définition d'une classe A :
- effectue des vérifications de typage.
- déclare le type des objets de la classe A un sous-type du type de l'interface I.

Cela permet par exemple, de voir tous les objets imprimables avec un type commun `Printable` et de leur envoyer des messages.

On peut donc écrire une méthode

```
void print (Printable p) { p#print; }
```

qui imprime les objets imprimables.

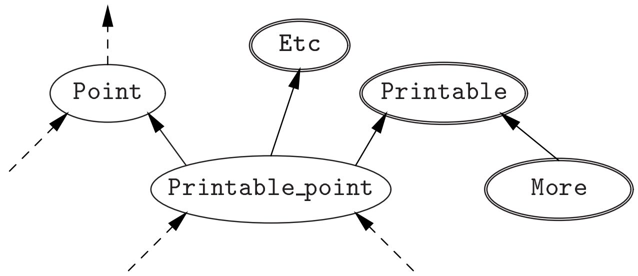
Slide 9

Interfaces et implémentations

La relation de sous-typage Une classe ne peut hériter que d'une seule classe, mais elle peut simultanément implémenter une ou plusieurs interfaces.

La relation d'héritage est un arbre, mais la relation de sous-typage est un graphe (treillis)

Slide 10



Le sous-typage

En java, le sous-typage est par nom.

– Une déclaration de classe `class A extends B implements` $B_1, \ldots B_k$ déclare un nouveau type `A`, et le déclare sous-type de `B` et de chaque B_i.
– Une déclaration d'interface `interface A extends` $B_1, \ldots B_k$ déclare un nouveau type `A`, et le déclare sous-type de chaque B_i.

La relation de sous-typage est fermée par reflexivité et transitivité.

Slide 11

Les casts

Le cast d'un objet change son type statique, mais ne change pas sa représentation (valeur dynamique).

Un cast se note `(A) (v)` où `A` est le type cible et `v` l'argument.

Lorsque le type statique de `v` est un sous-type de `A`, alors le cast est statique, et il réussit toujours.

Slide 12 Lorsque le type statique de `v` est un sur-type de `A`, alors il est possible que le type dynamique de `v` soit en fait plus précis et un sous-type de `A`. Le cast est autorisé, mais il nécessite un test à l'exécution qui peut échouer.

Tout `cast` qui peut réussir à l'exécution est permis.

Lorsqu'un cast `(cast)(v)` est rejeté à la compilation, il est toujours possible de le forcer par `(cast)((Object)(v))`, mais ce cast est certain de produire une erreur à l'exécution (sinon, il n'aurait pas été rejeté à la compilation) !

Les conversions

Un cast (conversions) effectue une conversion de type explicite.

Certaines conversions de types sont implicites. Par exemple, une affectation d'une valeur à une variable de type `A` effectue une conversion implicite vers le type `A` pourvu que la valeur soit un sous-type de `B`. Les conversions implicites ne peuvent pas échouer à l'exécution.

Slide 13 Les règles de conversion sont complexes (voir The Java Language Specification pour une description complète...)

(Voir exercise sur la surcharge)

Cast (exemple)

Slide 14

Il y a essentiellement trois types d'utilisation des `cast` :

– Les conversions statiques, pour changer le comportement d'une programme. En affaiblissant le type statique d'un objet vers le types des objets d'une classe parente, on change la résolution de la surcharge. L'objet se comporte alors comme s'il était de la classe parente.
– Les conversions dynamiques mais sûrs. Pour pallier une faiblesse du système de typage, il y a des situations où le type statique d'un objet est celui d'un classe parente de sa classe réelle. L'utilisateur peut le savoir (parce qu'il connait le code et peut faire un calcul mental) alors que le compilateur ne sait pas le vérifier statiquement (sans faire de calcul).
– Les tests de classes. Lorsque des objets de plusieurs classes sont mis dans un conteneur, à leur retrait, il faut parfois renforcer leur type à l'aide d'une conversion combinée avec un typecase.

Exercice 17 (Méthodes binaires) *On considère le programme suivant en Ocaml.*

Slide 15

```
class point x0 =
  object (this : 'mytype)
    val x : int = x0
    method getx = x
    method min (z : 'mytype) =
      if this#getx < z#getx then this else z
  end;;

class bipoint x0 y0 =
  object (this : 'mytype)
    inherit point x0 as super
    val y : int = y0
    method gety = y
    method min z =
      if this#gety < z#gety then this else super#min z
  end
```

```
let p = new bipoint 1 2;;
let q = new bipoint 2 1;;
let r = p#min q in r # gety;;
```

Écrire en Java le programme le plus proche possible. Commenter, et discuter éventuellement les différentes solutions. Réponse □

Slide 16

Typecase

La construction `v instanceof A` permet de tester si l'objet est d'une sous-classe de la classe `A`.

En fait `v instanceof A` retourne `true` si et seulement si l'expression `(A) (v)` ne produira pas d'exception.

De plus l'une ou l'autre expression sont rejetées par le compilateur dans exactement les mêmes conditions.

Slide 17

Deux utilisations typiques

- En conjonction avec une conversion, pour recouvrer le vrai type d'un objet qui a été perdu (en général suite au sous-typage).
- Seul, pour distinguer des objets de classes différentes (vues sous une même interface compatible.)

Les packages

Slide 18

Les packages sont un moyen de grouper plusieurs classes ensembles, en limitant la visibilité de certaines composantes à un package.

Les composantes des packages sont des classes, des interfaces, ou des sous-packages.

La fabrication et l'utilisation des packages dépend du système sous-jacent (conventions de nommage, de placement, et d'accès), mais le langage fixe les déclarations correspondantes dans le code source, ainsi que les règles de visibilité.

Les packages

Slide 19

Utilisation Pour accéder aux composantes `x`, `y` d'un package `P` on utilise la notation pointée `P.x`, `P.y`, *etc.*

La déclaration `import P.Q.x;` rend visible la déclaration `x` du package `P.Q` dans l'unité de compilation sans avoir à utiliser la notation `P.Q.x`.

La déclaration `import P.Q.*;` rend visible toutes les déclarations du package `P.Q` dans l'unité de compilation.

Fabrication La déclaration `package P.Q;` rend l'ensemble des déclarations qui suivent comme une partie du package `P.Q`.

Tout programme fait partie d'une unité de compilation nommée ou anonyme.

Les règles de visibilité

Classes, champs, méthodes et constructeurs peuvent être :
- `public`, *i.e.* visible partout ;
- non annoté, *i.e.* visible dans le package (nommé ou annomyme) auquel appartient l'unité de compilation.

De plus, champs, méthodes et constructeurs peuvent être :
- `private`, *i.e.* visible que dans la classe dans laquelle il est défini.
- `protected`, *i.e.* visible dans le package et dans les sous-classes de `C` (si la classe est publique).

Slide 20

La redéfinition dans une sous-classe d'une méthode d'une classe parente doit avoir une visibilité au moins aussi grande.

Contrainte Une classe `A` utilisée dans un autre fichier que celui où elle est définie doit être définie dans un fichier du nom `A.java`.

Type abstraction

Les règles de visibilité fournissent une forme faible de type abstrait.

En effet, contrairement à Ocaml, les composantes privées d'une classe `C` restent visibles dans la définition de la classe : en particulier dans le corps de la classe `C` un object de la classe `C` même autre this, expose ses composantes privées.

Slide 21

Dans tout autre classe (y compris une sous-classe de `C`) les composantes privées de C ne sont pas visibles. Ainsi, le type d'une classe est partiellement abtrait (les méthodes privées ne sont pas visible) en dehors de la classe qui le défini.

En Ocaml, la seule façon d'obtenir cet effet est d'utiliser le mécanisme d'abstraction des modules. La méthode est laissée publique mais son type est rendu abstrait.

Exercice 18 *On donne le programme suivant :*

```
class Codage {
```

```java
  private int code;
  Codage (int n) { code = n; }
  boolean decode (Codage z) { return (z.code == code); }
  }

 class Test_Codage {
  public static void main (String arg[]) {
  Codage p = new Codage (1);
  Codage q = new Codage (2);
  if (p.decode(q)) System.out.println("Please, enter");
  else System.out.println("Go away!");
  }
 }
```

Slide 22

Vérifier que la variable d'instance `code` *n'est pas accessible dans les objets codés.*

Donner un programme en Ocaml assurant le même niveau de sécurité. Réponse □

Packages (exemple)

Slide 23

P/A.java

```java
package P;
class A {
  private int x;
  public int y = x;
  A() { }
}
```

P/B.java

```java
package P;
public class B {
  public A a = new A();
  protected int m ()
    { return a.y; }
}
```

P/C.java

```java
package P;
public class C {
  B b = new B();
  int m () { return b.m(); }
}
```

User.java

```java
import P.B;
class User extends P.B {
  User () { m(); }
  B b = new B();
  int bm () { return b.m(); }
  int bay () { return b.a.y; }
}
```

Packages (exemple)

Visibilités

Classe	Importé	Publique	Package	Classe
A		A.y	A	A.x
B	A, A.y	B, B.a	B.m	
C	A, A.y, B, B.a, B.m	C	C.b, C.m	
User	P.B	User		

Slide 24

Errors

(1) La méthode `m` de la classe `B` est protégée dans `b.m`.

(2) Le type `A` de `b.a` n'est pas publique dans `b.a.y`.

La méthode `m` de la classe `User` est celle de la class `B`. Elle est visible dans `User` qui est une sous-classe de la classe `B` visible ; mais elle n'est pas visible dans un objet de la classe `B` elle-même.

Exercices

Reprendre quelques exercices Ocaml en Java...

Exercice 19 (Listes) *Écrire les listes sous forme de classe. Les spécialiser avec un constructeur `Append`.* □

Exercice 20 (Gestionnaire de fenêtres) *Reprendre le modèle maitre/esclave et le gestionnaire de fenetre.* □

Slide 25

6

Programmation par motif et motifs de programmation.

Slide 1

Cours	**Exercices**
1. Motif subjet/observateur 2. Protocoles couplés (Abonnements, Rappelle-moi, Relais) 3. Reconfiguration dynamique (Reconfiguration, Méthodes mutables, Crochets, Délégation) 4. Vues	1. Reconfiguration 2. Gestionnaire de fenêtres 3. Modèle multiplexé 4. Le général et son armée

Prologue

Slide 2

La programmation par motif est une notion suggestive et imprécise. Elle désigne le choix d'un protocole de communication ou d'assemblage entre différents types de composants.

Nous donnerons des exemples de motifs d'assemblage qui reviennent fréquemment. On pourrait parler simplement de techniques de programmation avec objets. Lorsqu'un motif particulier (ou une combinaison de plusieurs motifs) devient répétitif voire indispensable dans une application donnée, on peut parler de programmation par motif.

Il est souvent possible d'utiliser différents motifs pour résoudre un problème, mais le choix du motif retenu est rarement indifférent. Nous essayerons de comparer certains motifs entre eux dans des situations particulières.

Quelques exemples

Slide 3

Librairie de structures algébriques

Nous avons vu comment combiner la modularité apportée par les modules (générativité, abstraction) et par les classes (extension, liaison tardive) pour l'application à une librairie de calcul formel.

Une fois le style de programmation fixé, toutes les structures algébriques de la librairie devront suivre le même modèle.

Le protocole subjet/obervateur (voir ci-dessous), avec plusieurs variantes (volontaire ou autoritaire, à entrées multiples ou multiplexées).

De petites motifs qui reviennent fréquemment

- Les fonctions amis.
- Gestion d'événements.
- Abonnements à des services.

Le pattern sujet/observateur.

Présentation

Par exemple, il s'agit d'implémenter un gestionnaire de fenêtres.

Gestionnaire - Fenêtre
– les opérations sont réalisées par les fenêtres.
– les décisions sont prises par ou (ou plusieurs) gestionnaires.
L'implémentation doit rester extensible.

Slide 4

Le motif Subjet / Observateur est une généralisation :
– Les sujets exécutent les actions et rapportent aux observateurs
– Les observateurs décident et ordonnent aux sujets.
(Ici, le sujet est volontaire, i.e. il n'est pas obligé d'exécuter les ordres, mais une variante autoritaire serait similaire.)

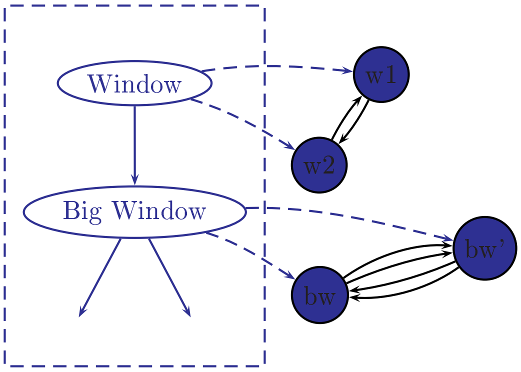
Le motif des listes est similaire, mais plus simple : `nil` *et* `cons` *sont des classes qui peuvent être raffinées,* e.g. *avec une méthode longueur ; de plus une nouvelle classe* `append` *peut être ajoutée. Cependant, la* *communication entre les différentes classes est moins intime que dans le motif subjet/observateur.*

Le motif du sujet/observateur est une généralisation des méthodes binaires où, cette fois-ci, ce sont des objets de classes différentes (et non de la même classe) qui interagissent entre eux.

Slide 5

Slide 6

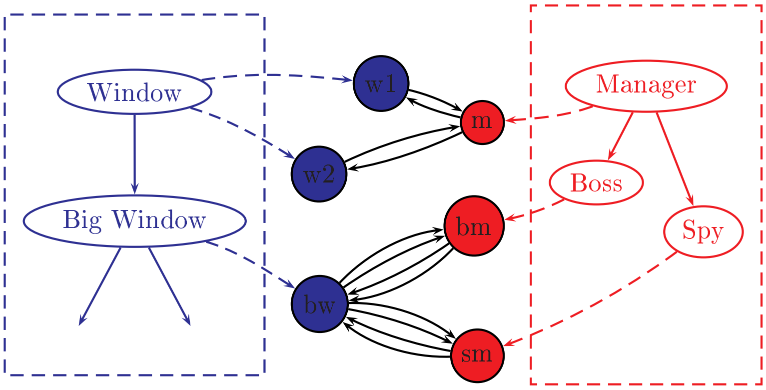
Héritage



L'héritage fonctionne bien, même avec des méthode binaire : chaque classe est paramétrique en le type de self.

Héritage de composants

Slide 7



La difficulté est d'assurer que la connection entre les objets des classes liées reste correcte et extensible dans les classes héritées.

La classe en bleu est paramértique en le type des objets en rouge et vice-versa.

Classes collaboratrices

Slide 8

De façon générale, deux classes travaillent ensemble à l'accomplissement d'une même tâche. Les objets de l'une et de l'autre s'appellent mutuellement, de façon intime, donc en partageant une certaine interface.

Le protocole ainsi établi entre deux classes doit pouvoir s'étendre en un protocole plus fin entre deux sous-classes tout en préservant la sûreté des échanges (les messages doivent toujours être bien compris).

Pour le protocole sujet/observateur, on fournit un protocole minimal permettant au sujet et à l'observateur de communiquer, sans préciser les valeurs échangées initialement.

Par la suite, on étendra ce modèle par héritage à des situations concrètes en précisant les informations échangées.

Le sujet

C'est le sujet qui maintient à jour l'ensemble des observateurs à qui il devra rendre compte. Il utilise pour cela une variable d'instance `observateurs` et une méthode `ajoute` pour gérér ses abonnés, et une méthode `envoie` pour faire suivre un message à l'ensemble des abonnés.

Slide 9

```
class ['obs] sujet =
  object (self)
    val mutable observateurs = ([]: 'obs list)
    method ajoute m = observateurs <- m :: observateurs
    method envoie (message : 'obs -> 'mytype -> unit) =
      List.iter (fun obs -> message obs self) observateurs
  end;;
```

Le modèle est coopératif : ici c'est l'observateur qui maintient la liste de ses maîtres. On peut facilement inverser le modèle.

L'observateur

Dans la version dépouillée, l'observateur n'a rien à gérer et il n'y a pas encore de communication : c'est une classe vide.

```
class virtual ['subjet] observateur = object end;;
```

Slide 10

Le gestionnaire de fenêtre

Slide 11

Les fenêtres héritent du modèle du sujet :

```
class ['obs] fenêtre =
  object (self)
    inherit ['obs] sujet
    val mutable position = 0
    method déplace d =
      position <- position + d;
      self#envoie (fun x -> x#déplacé)
    method dessine =
      Printf.printf "{Position = %d}" position;
  end;;
```

La méthode `déplace` exécute le déplacement et rend compte d'un déplacement en envoyant par convention le message `déplacé` à l'observateur. La fenêtre a également une méthode dessine.

Le gestionnaire de fenêtre (suite)

Slide 12

Le gestionnaire hérite du modèle de l'`observateur` :

```
class ['sujet] gestionnaire =
  object
    inherit ['sujet] observateur
    method déplacé (s : 'sujet) : unit = s # dessine
  end ;;
```

La méthode `déplacé` de l'observateur prend acte du déplacement et demande à l'objet (reçu en argument) de se redessiner.

Raffinement du gestionnaire de fenêtre

On teste d'abord le modèle :

```
let fenêtre = new fenêtre in
fenêtre#ajoute (new gestionnaire); fenêtre#déplace 1;;
```

{Position = 1}− : unit = ()

Slide 13

Le classes `fenêtre` et `gestionnaire` peuvent être raffinées par héritage, indépendemment l'une de l'autre. Par exemple, on pourra :

– ajouter des méthodes `retaille` et `devant` aux fenêtres.
– améliorer la méthode dessine...
– faire prendre en compte de nouveaux messages au gestionnaire.

Le contremaître surveille

On peut aussi ajouter un nouveau maître indépendant, par exemple qui se contente de tracer les opérations effectuées :

Slide 14

```
class ['sujet] inspecteur =
  object
    inherit ['sujet] observateur
    method retaillé (s: 'sujet) = print_string "<R>"
    method déplacé (s: 'sujet) = print_string "<D>"
  end;;

let f = new fenêtre in
  f#ajoute (new gestionnaire);
  f#ajoute (new inspecteur :> 'a gestionnaire);
  f#déplace 2;;
```

<D>{Position = 2}− : unit = ()

Exercice 21 (Gestionnaire de fenêtres) *Dériver une classe* *`grande_fenêtre` qui implémente une méthode `retaille` (ce qui implique aussi de redéfinir la méthode `dessine` pour afficher la taille) qui rend compte en appelant la méthode `retaillée` de l'observateur. On ajoutera également une méthode `en_avant` pour placer une fenêtre sur le devant (représenté par un booléen).* Réponse

Dériver une classe `grand_gestionnaire` capable de gérer les grandes fenêtres (par exemple, à la récep Réponse

Étendre la classe `inspecteur` en `big_brother` qui trace tous les

Slide 15 *ordres.* Réponse □

Exercice 22 (Modèle multiplexé) *Un problème est de ne pas pouvoir écrire pouvoir considérer l'inspecteur comme un observateur pour les grandes fenêtres, alors qu'il pourrait se contenter d'ignorer les messages qu'il ne connnait pas.*

Pour cela on peut choisir un autre protocole, dans les messages sont mutiplexés, i.e. *ils passent tous par le même canal ; pour cela, ils sont taggés en entrée, par exemple en utilisant les variantes), et on peut leur associer un comportement par défaut consistant à ne rien faire.*

Reprendre le sujet/observateur selon ce modèle. □

Exercice 23 *Utiliser une variante du motif sujet/observateur pour modéliser l'exemple du "général et du soldat".* □

Slide 16

Protocoles couplés.

Un motif, à la loupe

Slide 17

Dans la plupart des cas, on se retrouve dans une situation analogue au client/serveur, maître/esclave, *etc.*, *i.e.* deux classes doivent s'appeler l'une et l'autre en se passant de l'information, des messages, *etc.* tout en restant extensibles...

Dans le modèle maître/esclave, nous avons vu plusieurs solutions :

- Multiplexer les messages sous forme d'un type variante pour les faire passer par le même canal.
- Séparer les messages, en utilisant une paire de noms (émetteur/récepteur) par type de message.

En pratique, on retrouve souvent ce dilemme, et il est important de choisir la bonne combinaison.

Événements clavier et souris

Multiplex

Tous les événements sont séquentialisés et transmis de la même façon. Le récepteur trie et traite les événements de façon appropriée.

Séparation

Slide 18 Le gestionnaire trie les événements clavier/souris selon les différents types (ou combinaison) d'événements et appelle directement des méthodes dédiées.

Événements clavier et souris (comparaison)

- Le multiplex nécessite de figer dès le départ l'ensemble des événements, à monis d'utiliser des types variantes.
  Est-ce un problème dans le cas traité ? (cela dépend de l'application considérée)
- Le multiplex traite tous les événements de la même façon, le code est donc bien partagé.

Slide 19

- La séparation paraît plus souple : chaque type d'événement est associé à une méthode dédiée et peut donc être traité de façon appropriée.
- En contrepartie, le nombre de méthodes peut augmenter rapidement, beaucoup étant semblables.
- La multiplication du nombre de méthodes rend souvent le protocole plus confus et l'héritage plus contraignant (tous les composants doivent définir toutes les méthodes).

Événements clavier OU souris

On va souvent choisir une solution mixte, partitionant les événements en différents groupes, traités par des méthodes différentes pour chaque groupe mais par la même méthode au sein d'un même groupe.

Remarque : si besoin, on peut fabriquer des événements composites au niveau du gestionnaire (*e.g.* le double clique).

Slide 20

Les abonnements

Aucune des deux solutions précédentes ne permet de modifier dynamiquement la gestion des événements. Pour changer les événements reçus, il faut le faire au niveau de la classe par héritage.

On peut abstraire le traitement des événements par un système d'abonnement. Le client s'abonne à des services auprès du serveur. L'abonnement peut être effectué et annulé dynamiquement.

Slide 21

En contrepartie, les services fournies doivent être homogènes (opérer sur le même type).

Abonnement exemple

L'abonnement le plus simple est un service (dont les messages sont envoyés par effet de bord).

```
let remove c = List.find_all (fun (x,_) -> x <> c)

class ['c] serveur = object
  val mutable clientèle : ('c * ('c -> string -> unit)) list = []
  method enregistre c f = clientèle <- (c, f) :: clientèle
  method désabonne c = clientèle <- remove c clientèle
  method send m = List.iter (function c, f -> f c m) clientèle
end

class ['serveur] client (s : 'a #serveur) = object (self : 'a)
  method enregistre f = s#enregistre self f
  method désabonne = s#désabonne self
  method attend_le_résultat =
    self#enregistre (fun z s -> z#imprime_le_résultat s)
  method imprime_le_résultat s = print_string s;
end;;
```

Slide 22

Abonnement (test)

```
let s = new serveur;;
let c = new client s;;
c # attend_le_résultat;;
s # send "Bonjour, "; s # send "tout le monde!";;
c # désabonne;;
s # send "Toc! toc!"; s # send "Y'a quelqu'un?";;
```

Slide 23

À noter en passant, l'emploi d'un relais : l'utilisateur appelle `self#désabonne` qui en retour appelle `s#désabonne f` de telle façon que l'utilisateur n'ait pas à connaître (ou rappeler) le serveur `s`.

Inconvénient

Les messages sont homogènes, ce qui ne permet pas de gérer plusieurs clientèles.

Messages hétérogènes

On peut homogénéiser les messages de la façon suivante : un abonnement est une paire de type `client * client -> int -> unit` qui expose le type du client et le type du message (`int` ici). Le type du client peut être caché en appliquant la fonction à l'argument de façon retardée :

```
val mutable clientèle : (string -> unit) list = []
method enregistre c f =
  clientèle <- (fun z -> f c z) :: clientèle
```

Slide 24

Il est alors possible de gérer des abonnements de clients hétérogènes.
Mieux encore :

```
val mutable clientèle : (string -> unit) list = []
method enregistre fc = clientèle <- fc :: clientèle
```

Pour pouvoir se désabonner, on peut maintenir l'objet comme première composante, mais de type `< >`.

Messages hétérogènes

Slide 25

```
type obj = < >

class serveur = object (self)
  val mutable clientèle : (obj * (int -> unit)) list = []
  method désabonne f = clientèle <- remove f clientèle
  method enregistre (f : obj) fc = clientèle <- (f, fc) :: clientèle
  method send m = List.iter (function _, fc -> fc m) clientèle
end

class ['serveur] client (s : #serveur) = object (self)
  constraint 'serveur = #serveur
  method enregistre f = s#enregistre (self :> obj) (f self)
  method désabonne = s#désabonne (self :> obj)
  method attend_le_résultat =
    self#enregistre (fun z s -> z#imprime_le_résultat s; z#désabonne)
  method imprime_le_résultat s = print_int s
end;;
```

Messages hétérogènes

L'utilisation est la même, mais les types sont moins exposés.

```
let s = new serveur in let c = new client s in
c # attend_le_résultat; s # send 1; s # send 2;;
```

1− : unit = ()

On peut maintenant soumettre des abonnements de clients hétérogènes.

Slide 26 Il ne s'agit pas d'un affaiblissement du typage, l'exposition du type de client au serveur n'étant pas nécessaire. Une autre solution aurait été d'utiliser des types existentiels :

Types existentiels

En fait, la paire `(f, c)` a le type $\exists\alpha.(\alpha \times (\alpha \rightarrow \texttt{int} \rightarrow \texttt{unit}))$ dans lequel la variable α n'est pas libre (elle est donc cachée et on parle de type abstrait ou existentiel).

Rappelle-moi (callbacks)

L'exemple précédent est une forme dite de *callback*.

On donne au serveur une fonction permettant de rappeler le client. Typiquement, un callback est une fonction de type `unit -> unit` mais comportant dans sa fermeture des effets de bords permettant d'informer le client d'événements détectés par le serveur.

Slide 27 Un rappelle-moi est donc un moyen de geler une évaluation dans un objet (A), on dit que l'on forme un glaçon (G), et de passer (G) à un autre objet (B). C'est alors (B) qui a la responsabilité de réveiller (G) et de relancer son exécution.

Rappelle-moi (force)

Slide 28

La force de cette technique réside dans le fait d'encapsuler le calcul dans une fermeture. Cela augmente à la fois deux propriétés souvent antagonistes :

– la sécurité en limitant le risque d'intrusion
– la flexibilité en rendant la plupart des informations non visibles (élimination des contraintes de typage).

Attention La force en fait sa faiblesse...

Les callbacks, en général de type `unit -> unit`, peuvent être accidentellement interchangés. La sûreté (différent de sécurité) est donc diminuée.

Rappelle-moi (faiblesse)

Slide 29

Inversement, la faiblesse de cette technique est que contrairement aux structures de données qui peuvent être analysées et interprétées (*i.e.* les opérations possibles ne sont pas contenues dans les données), les évaluations gelées ne peuvent qu'être réveillées.

Voir le contournement de cette difficulté dans l'exemple des abonnements pour implanter le désabonnement. On ne peut même pas tester l'égalité de deux continuations, sinon leur égalité physique, car ce sont des fonctions.

Rappelle-moi (callbacks)

Slide 30

Le mécanisme de "rappelle-moi" est très utilisé dans un monde concurrent. En effet, lorsque plusieurs processus contribuent ensemble à l'évaluation, le calcul n'est plus un flux linéaire de fonctions qui se passent des arguments et retournent des résultats.

Un processus doit fréquemment geler son exécution, et attendre qu'un autre lui envoie un événement pour continuer. On retrouve un mécanisme un peu analogue à une évaluation gelée passée à quelqu'un d'autre qui doit en assurer le réveil...

Les exécutions gelées sont aussi appelées des *continuations*.

Rappelle-moi (exemple)

Slide 31

On reprend l'exemple du client-serveur dans le contexte d'un gestionnaire d'événements.

Le client indique au serveur qu'il veut recevoir tel événement, et passe simultanément la façon de se faire rappeler. Cet exemple est combiné avec un mécanisme de table, ici délocalisée.

Les mécanismes d'abonnements ont aussi presque toujours besoin d'un rappelle-moi pour retourner les messages.

```
class serveur = object (self)
  val mutable filtres : (char * (unit -> unit)) list = []
  method écoute c f = filtres <- (c, f)::filtres
  method étape = List.assoc (input_char stdin) filtres ()
  method boucle = while true do self#étape done
end;;
```

Rappelle-moi (exemple)

Slide 32

```
class client (s: #serveur) = object
  val mutable position = 0
  method position = print_int position
  method avance = position <- succ position
  method recule = position <- pred position
  method si_faire c f = s#écoute c f
end;;
let s = new serveur in let c = new client s in
c # si_faire 'a' (fun() -> c#avance);
c # si_faire 'r' (fun() -> c#recule);
c # si_faire 'p' (fun() -> c#position);
s # boucle;;
apaaaaaaaappp.
```

1999Uncaught exception: Not_found.

Les relais

Slide 33

Les méthodes "`enregistre`" et "`désabonne`" du client sont des exemples de relais : Elles font suivre les messages adressés au client vers des messages associés du serveur.

```
  method enregistre f = s#enregistre self f
  method désabonne = s#désabonne self
```

Encapsulation

Le relais permettent au client de ne pas exposer certains paramètres (ici le serveur) qui restent encapsulés dans le client.

Indépendemment, un appel distant (en dehors de la classe) peut ne pas connaître le nom du serveur. Le message est alors envoyé au client, qui le complète avec éventuellement des informations seulement connues par le client et le retourne au serveur.

Les relais (liaison tardive)

Sobriété Un relais évite de rappeler certains paramètres à chaque appel. Lorsque ces appels sont dans la classe elle-même, alors le relais joue le rôle d'une liaison locale visible dans plusieurs méthodes (on utilise alors un relais privé).

Slide 34

Modularité Des relais peuvent aussi être utilisés pour leur mécanisme de liaison tardive, permettant de modifier le comportement dans une sous-classe. Par exemple, la méthode `désabonne` peut faire des mises à jour, tracer, ou clore d'autres opérations.

```
class ['serveur] client' s = object (self)
   inherit ['serveur] client s as sup
   method désabonne = print_newline(); sup#désabonne self
  end;;
```

Les relais (collecteur)

Le relais peut aussi être utilisé pour faire automatiquement un travail de collecte de détail, visitant plusieurs objets qui lui fournissent des informations personnelles, avant d'atteindre sa cible finale et de s'exécuter.

Slide 35

Modularité (statique) et reconfiguration (dynamique).

Flexibilité

La modularité fait référence à la possibilité de réutiliser certains composants en les adaptant ou en les spécialisant. Cette extensibilité, a priori statique, revient donc a définir de nouveaux composants à partir d'anciens.

Slide 36

La reconfiguration permet aussi d'adapter des composants en changeant leurs propriétés ou leurs comportements. Cependant à la différence de la modularité, la reconfiguration est dynamique.

Par son caractère dynamique la reconfiguration est plus expressive que la modularité. En contrepartie, elle ne peut se faire que de façon limitée, en particulier par le typage : la reconfiguration ne peut pas changer les types.

Extensibilité

Slide 37

Les exemples d'extensibilités sont bien connus. Elle repose essentiellement sur l'héritage dans les classes et l'application de foncteurs dans les modules.

Par exemple, à partir d'une classe `figure` et d'un comportement de bouton, on fabrique une classe bouton. Ou encore, on personnalise une classe figure en lui ajoutant des menus locaux, *etc.*

Reconfiguration

Slide 38

Dans sa forme la plus simple, la reconfiguration peut ne changer que des paramètres tels que la couleur, l'orientation, les droits *etc.*

Par exemple, on peut ainsi dire ou interdire à un composant de se "retailler automatiquement" ou d'accepter l'ajout ou le retrait de nouveaux éléments. Ces exemples ne font souvent que changer des variables d'états (drapeaux) du composant.

Dans une forme plus expressive, la reconfiguration peut aussi modifier le comportement de certaines méthodes souvent en ajoutant une action au comportement d'origine.

Enfin dans une forme plus poussée, un objet va complètement se restructurer et se comporter d'une toute autre façon sans pour autant changer d'identité.

Reconfiguration

Slide 39

En pratique, le passage d'une forme à l'autre est souvent clair selon le contexte, mais en théorique cette progression est continue. En particulier, les méthodes de reconfiguration s'appuient presqu'exclusivement sur les effets de bords donc la modification de champs mutables.

Grossièrement on pourrait classer les différents types de reconfiguration en fonction du contenu des variables mutables : données, fonctions, ou objets.

La reconfiguration revient donc à simuler dynamiquement l'effet de l'héritage statique. Cette simulation est souvent approchée et limitée. Elle se fait en général par une indirection, souvent par effet de bord, mais parfois de façon fonctionnelle.

(Nous laissons de coté la reconfiguration simple qui s'implémente évidemment par un champ mutable.)

Méthodes mutables

Slide 40

Les méthodes mutables n'existent pas, mais cet effet peut être obtenu simplement en faisant exécuter par la méthode le contenu d'une variable mutable.

```
class ['a] action =
  object (self : 'z)
    val mutable action : 'z -> 'a =
      fun s -> raise (Failure "No action")
    method action = action self
    method set_action f = action <- f
  end;;
```

En pratique, la classe a évidemment d'autres méthodes statiques et seulement quelques méthodes particulières que l'on veut rendre mutables et pour lesquellles on définit un triplet formé d'une variable d'instance mutable, d'une méthode d'invocation et d'une méthode d'affectation.

Héritage des méthodes mutables

On peut aussi simuler l'héritage, à condition de fixer un protocole d'appel (ici un seul argument)

```
class ['a] super_action = object
  inherit ['a] action
  method post_action f =
    let super = action in
    action <- (fun self z -> f self (super self z))
  method pre_action f =
    let super = action in
    action <- (fun self z -> super self (f self z))
end;;
```

Slide 41

Généralisation et spécialisation On peut facilement remplacer l'action par une liste d'actions à composer (ou à itérer dans le cas où elles ne retournent rien).

Les crochets (hooks)

Les crochets (hooks) permettent de prévoir des emplacements avec des comportements par défaut qui pourront être changés ultérieurement. C'est une variation sur les méthodes mutables.

Emacs est l'exemple typique d'utilisation de cette technique : la plupart des fonctions peuvent être personnalisées en accrochant des fonctions de personnalisation aux modes fournis pas défaut.

Slide 42

Les crochets sont en fait un concept plus général qui ne s'applique pas qu'à l'orienté objets. Les méthodes mutables sont un moyen, parmi d'autres, de l'implémenter, y compris en présence d'objets.

Comme pour les méthodes mutables, il faut prévoir un emplacement modifiable, pour pouvoir augmenter le comportement ultérieurement, souvent initialisé avec du code vide.

Les tables

Les tables sont une généralisation des méthodes mutables. Cela revient à multiplexer plusieurs méthodes. Pour cela, on passe un paramètre particulier utiliser comme clé dans une table de méthodes (peu importe la représentation de la table).

Typiquement, le traitement des événements peut se faire par une table d'association utilisant le type de l'événement comme clé.

Slide 43

Par rapport à l'approche statique, le typage est à la fois contraint (tous les messages ont le même type) et relâché (il y a un message par défaut pour les clés non définies, en général *"ne rien faire"*, afin d'éviter l'erreur *"message non compris"*).

Cette approche permet de modifier la réaction à certains événements. L'ajout et le retrait reviennent à une modification du comportement par défaut.

Exercice 24 *On peut imaginer le de pousser à fond le mécanisme de reconfiguration et traiter tous les messages comme des messages dynamiques.*

1. *Quel serait le problème principal rencontré ?*
2. *Décrire deux façons de le contourner.*
3. *Illustrer l'une des deux sur un exemple simple tels que les points colorés.*

□

Slide 44

La délégation

Slide 45

Cela revient à déléguer certaines méthodes aux méthodes d'un objet d'une autre classe. La création d'un objet (a) d'une classe (A) provoque la création d'un objet (b) d'une autre classe (B).

L'état de l'objet (a) comporte indirectement l'état de l'objet (b). On peut alors adresser un message à (b) en donnant accès directement à (b) au travers de (a) ou en relayant certaines méthodes de (b) dans (a).

La délégation est un style de programmation avec objet utilisé indépendemment de la reconfiguration, à la place de l'héritage. Il facilite certaines reconfigurations.

La délégation (exemple simplifié)

Slide 46

Contexte

```
class texte s0 =
  object
    val mutable s = s0 method s = s
    method imprime = print_string s
  end
class point x0 =
  object
    val mutable x = x0 method x = x
    method imprime = Printf.printf "<%d>" x
  end;;
```

But

Écrire une classe `point_annoté` à partir des classes précédentes.

La délégation (exemple simplifié)

Slide 47

Version avec héritage

```
class point_annoté x0 s0 =
  object
    inherit point x0 as point
    inherit texte s0 as texte
    method imprime = point#imprime; texte#imprime
  end;;
```

Version avec délégation :

```
class point_annoté x0 s0 =
  object
    val point = new point x0 method point = point
    val texte = new texte s0 method texte = texte
    method imprime = point#imprime; texte#imprime
  end;;
```

Délégation et héritage

Slide 48

Héritage

- Les deux classes parentes sont aplaties en une seule qui conserve la liaison tardive et permet de spécialiser toutes les méthodes.
- L'accès privilégié aux variables d'instance `x` et `s` est normal.

Délégation

- Les méthodes peuvent avoir le même nom dans les deux classes parentes. (on peut former une classe de bipoints avec deux points).
- On peut maintenant facilement reconfigurer un objet de la classe `point_annoté`, en changeant par exemple l'objet `texte` par un objet `texte_éditable` (il suffit de rendre mutables les champs `point` et `texte`.

Délégation et héritage

Les points forts d'une approche sont les points faibles de l'autre.

En particulier, dans la délégation, la classe héritée n'a pas accès aux variables des classes parentes ; cela amène souvent à rendre visibles des champs ou des méthodes qui devraient logiquement rester privées (visibles seulement dans une sous-classe).

Slide 49

Inversement, il n'est pas possible d'avoir deux méthodes différentes portant le même nom dans la version avec héritage.

La version avec héritage est donc plus modulaire (bien que le dernier point soit une limitation). La délégation pallie à l'absence d'un mécanisme de vue.

Les vues

Les vues sont des interfaces (dictionnaires) entre les noms de méthodes et leurs implémentation. Un objet peut alors avoir plusieurs implémentation de la même méthode mais dans des vues différentes. Les vues conservent le mécanisme de la liaison tardive (l'espace des noms est hiérarchisée, mais les implémentations sont aplaties).

Slide 50

Malheureusement... les langages à objets existant n'offrent pas encore de mécanismes de vues (suffisamment sophistiqués).

Objets sans états...

Voir combinaison des modules et des objets.

C'est une utilisation atypique mais expressive des objets pour bénéficier de l'héritage et du mécanisme de la liaison tardive que n'offrent pas les structures.

C'est aussi d'une certaine façon un palliatif à l'absence de mixins.

Slide 51

7

Le modèle à enregistrement.

Slide 1

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

Prélude

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*.

Slide 2

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

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.

Slide 3

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 :

$e ::= x$	Variable (ou constante)
$\mid c$	Constante (ou primitive)
$\mid$ `fun` $(x)\, e$	Fonction
$\mid e_1\ e_2$	Application

Slide 4

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

Réduction

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 `fun` $(x)\, e$ et les constantes c.

Slide 5

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 β-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 δ-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 δ.
  (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

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

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

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

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

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

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

Exercice 25 *Réduire le terme*
$(\texttt{fun }(z)\,\texttt{fun }(t)\,\texttt{fun }(f)\,(z\ t)\ f)\ (\texttt{fun }(x)\,\texttt{fun }(y)\,x)\ 1\ 2.$ Réponse
Par quel terme peut-on remplacer $(\texttt{fun }(x)\,\texttt{fun }(y)\,x)$ *pour que le résultat final soit* 2 *? Commenter.* Réponse □

Slide 9

Exemple (suite)

Exercice 26 *Donner un exemple de programme erroné (qui ne se réduit pas sans pour autant être une valeur).* Réponse
Donner un exemple de programme qui boucle (qui se réduit indéfiniment). Réponse □

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

Slide 13

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.

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 ℓ dans p avec la définition $\varsigma\,(x)\, e$.

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

Exemple (dans le calcul avec fonctions)

Slide 16

Un point (approche fonctionnel) :

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

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)

Slide 17

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

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

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

Sémantique opérationnelle

Slide 18

Les valeurs sont les objets.

Réduction élémentaire

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

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

Contextes d'évaluation (évaluation en tête)

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

Exemple d'évaluation

Slide 19

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

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

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

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

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

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

$\boxed{\langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h\,\rangle.\mathtt{geth}} \longrightarrow$

$(x.h)\{x \leftarrow \langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h\,\rangle\} \equiv$

$\boxed{\langle\, h = 3; \mathtt{geth} = \varsigma\,(x)\,x.h\,\rangle.h} \longrightarrow$

3

Exercice 27 (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 :*

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

Réponse

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.)* Réponse □

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 :

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

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 :

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

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

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

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

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

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 :

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

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

Slide 24

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

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).

Slide 25

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
$\mid \{\, x \texttt{ with } \ell = e \,\}$	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

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

Contextes d'évaluation

Slide 28

$E ::= \ldots$	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 $(l_i = v_i)_{i \in I}$)

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

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

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

Exercice 28 (Enregistrements extensibles finis) *Dans cet exercice, on veut faire découvrir le mécanisme des enregistrements extensibles. On se limite à un nombre fini de champs.*

Slide 29

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 }` *.* Réponse

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) ?* Réponse

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.* Réponse

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* $\{ \{ m \}$ `with` $\ell = v \}$*) ? Définir les valeurs correpondantes pour le champ* `b` *ainsi que la valeur correspondant à l'enregistrement vide.* Réponse

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

Slide 30

Codage des objets (non typé)

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 :

Slide 31

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

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.

Exercice 29 (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` : Réponse

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

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. Réponse

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)

Slide 36

Distinguer objets et prototypes

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

Slide 37

Utilisation de vues

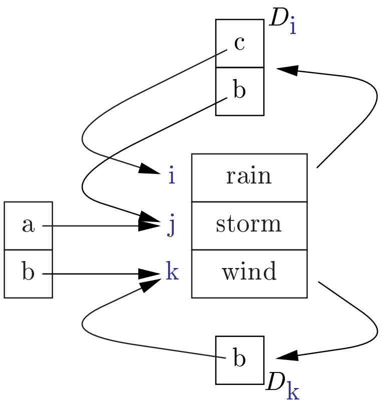
Elles permettent de renommer les noms de méthodes et ainsi de les dissocier de leur implémentation. Un même nom peut avoir plusieurs implémentations. On peut oublier un nom, et le redéfinir, mais avec une nouvelle implémentation (sans écraser l'implémentation précédente).

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_{i} et D_{k} sont des dictionnaires internes.

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

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

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

8

Le modèle à surcharge.

Slide 1

Cours	**Exercices**
1. Principe 2. Exemples 3. Multi-méthodes 4. Difficultés 5. Résolution 6. Cohérence 7. Encapsulation 8. Le langage Nice 9. Summary	1. Nice 2. Listes

Modèle à enregistrement (rappel)

Slide 2

Le modèle à enregistrements est le plus fréquent.
- Les objets sont des enregistrements
- Les variables d'instance et les méthodes sont des champs.
- Les messages sont des étiquettes.
- L'envoi de message est une auto-application : extraction du message de l'objet et application à l'objet
- Les classes sont des enregistrements (prototypes)
- L'héritage est l'extension des enregistrements

Le modèle à enregistrements permet aussi l'encapsulation, *i.e.* de cacher (par abstraction de type, sous-typage ou par limitation de la portée) les variables d'instance.

C'est le modèle du langage Ocaml, mais aussi des langages Java, C++, etc.

Modèle à surcharge (principe)

Slide 3

Dans le modèle à surcharge, les méthodes ne sont pas portées par les objets, mais définies en dehors des objets comme les fonctions dans la programmation traditionnelle.

Un message est une fonction surchargée, parfois appelée une fonction générique. Un message à donc plusieurs implémentations fonctionnant définies pour des objets de classes différentes.

L'envoi d'un message est l'application d'une fonction surchargée. Le mécanisme de résolution de la surcharge sélectionne *dynamiquement* l'implémentation la plus précise en fonction du destinataire, de sa position dans la hiérarchie des classes et de l'ensemble des définitions enregistrées.

Modèle à surcharge (principe)

Slide 4

En résumé :
- Les objets sont des enregistrements mais ne décrivant que leur état interne.
- Les variables d'instance sont des champs.
- Les méthodes sont des fonctions ordinaires.
- Les messages sont des fonctions surchargées.
- L'envoi d'un message est l'application d'une fonction surchargée à le (ou les) objet(s) destinataire(s).

Dans sa version de base, le modèle à surcharge ne permet pas l'encapsulation. (Mais on peut ajouter un mécanisme d'encapsulation au niveau de modules.)

C'est l'approche suivie par les langages CLOS, les fonctions génériques de Scheme, le langage Jazz ou Nice, le langage jouet KOOL. le langage Cecil.

Exemple (pseudo-code)

Slide 5

NB : ce code à la syntaxe "Ocaml" n'est pas du code Ocaml exécutable.

Classes On ne définit que les variables d'instance. Les classes peuvent hériter de classes parentes, dans ce cas, elles héritent de leurs variables d'instances.

```
class point = object val mutable x = 0 end
class cpoint c = object inherit point val couleur = c end
```

Méthodes La classe du receveur est précisé avec la notation `@` :

```
method getx (p @ point) = p.x
method print (p @ point) = print_int (getx p)
method setx (p @ point) y = p.x <- y
method getc (p @ cpoint) = p.c
```

Exemples (pseudo-code)

Création des objets

C'est comme la création d'une structure de donnée.

```
let p = new point;;
let cp = new cpoint "noir";;
```

Slide 6

Envoi d'un message

C'est l'application d'une fonction surchargée à un objet.

```
getx p, getc cp, getx cp;;
```

getc p;;

Mécanisme de résolution

La résolution est dynamique

Les objets portent donc nécessairement en plus de leur variables d'instance une étiquette indiquant leur classe.

Slide 7

Lorsqu'il existe une implémentation pour cette classe, c'est elle qui est choisie. Sinon, on cherche une implémentation de la méthode dans une classe parente la plus proche.

Résolution (super)

Slide 8

On peut modifier le mécanisme de résolution (utiliser la méthode d'une classe parente (super) même s'il y a une définition locale) en utilisant la notation `@`.

```
method print (p @ cpoint) =
  print (p @ point); print_string (getc p)
```

Cela revient à changer le type dynamique de `p`. (C'est un mécanisme entièrement différent du cast en Java qui ne change jamais la représentation des objets.)

(Une ancienne définition peut toujours être atteinte en le mentionnant explicitement)

```
print p;;
```

0noir;;

```
print (p @ point);;
```

0

Multi-méthodes

Slide 9

C'est une généralisation de l'approche précédente permettant d'envoyer un message à plusieurs récepteurs simultanément.

La résolution du message prend en compte tous les destinataires (on parle de *multiple dispatch*) pour choisir l'implémentation la plus précise.

Les langages CLOS, Jazz, Nice, Cecil ont des multi-méthodes.

Exemples (multi-méthodes)

Slide 10

On ajoute une méthode `equal` pour tester l'égalité sur les points. On définit un comportement par défaut pour les paires de points et un comportement spécial pour les paires de points colorés.

```
method equal (p @ point, q @ point) = p.x = q.x
method equal (p @ cpoint, q @ cpoint) =
  equal (p@point, q@point) && p.c = q.c
```

L'égalité d'un point et d'un point coloré n'étant pas définie, elle est traitée comme l'égalité de deux points.

```
equal p cp;;
```

true

Les deux arguments jouent ici un rôle parfaitement symétrique, contrairement au modèle à enregistrements.

Approche mixte

Slide 11

C'est une combinaison du paradigme traditionnel (approche à enregistrements) et des multi-méthodes. Comme dans le modèle à enregistrements, les méthodes sont toujours spécialisées par rapport à un objet, mais ce sont ensuite des fonctions surchargées qui comme les multi-méthodes résolvent le messages sur les arguments supplémentaires, pris simultanément.

Le langage Cecil est le représentant typique de cette catégorie.

Faux-amis

Java a l'apparence d'une approche mixte, car les méthodes sont surchargées. Cependant en Java la surcharge est résolue *statiquement*, donc on ne peut pas parler de multi-méthodes (La surcharge statique est une convenance de notation, mais n'augmente pas l'expressivité).

Codage du modèle à enregistrements

Si l'on oublie le problème de l'encapsulation de l'état interne, il est facile de coder le modèle à enregistrements dans le modèle à surcharge.

Il suffit de
– définir les classes comme précédemment, mais sans les méthodes
– définir les méthodes immédiatement après les définitions de classes.
Le modèle à surcharge est donc plus expressif que le modèle à enregistrements.

Slide 12

En contrepartie, et malgré les apparences, il est aussi beaucoup plus compliqué...

Difficultés du modèle à surcharge

Le modèle implicite précédent cache un certain nombre de difficultés. En particulier,

1. les méthodes peuvent être définies récursivement,
2. le mécanisme de liaison tardive implique une forme de récursion ouverte,
3. le mécanisme de résolution.
4. l'encapsulation entre en conflit avec la définition des méthodes en dehors des classes.

Slide 13

Les deux premières difficultés sont présentes également dans le modèle à enregistrements, où elle sont résolues classe par classe. Ces difficultés sont augmentées ici par la nécessité de traiter le problème globalement. C'est la source des difficultés de typage et de compilation, la compilation séparée étant très difficile.

Le mécanisme de résolution est un problème nouveau.

Restriction sur la formation des messages

Un message est une collection d'implémentations. Il existe deux conditions sur les message pour qu'ils soient bien former.

- Une condition de fermeture pour assurer que la résolution des messages est toujours possible de façon non-ambigü.
- Une condition de cohérence pour assurer que le typage statique est cohérent avec la résolution dynamique.

Slide 14

Mécanisme de résolution

Pour expliquer le mécanisme de résolution, on peut se ramener à un petit calcul avec surcharge (voir le livre de Castagna), mais la présentation reste complexe.

En particulier, il faut faire intervenir le typage dans le mécanisme de résolution qui en dépend.

Slide 15 Nous nous contentons ici d'une présentation informelle des problèmes et des solutions au mécanisme de résolution.

Résolution (cas simple)

En l'absence de multi-méthodes et d'héritage multiple, la résolution est plus simple, car une classe a au plus une classe parente.

Slide 16

Un message m est une liste de méthodes m_i de domaines c_i disjoints (une définition de même domaine qu'une définition existence cache l'ancienne). Lorsqu'il est appliqué à un objet d'une classe c, il suffit de trouver le plus petit élément parmi les c_i dont c est un sous-type (ce plus petit élément est bien défini).

Résolution (cas général)

La résolution dans le cas des multi-méthodes ou de l'héritage multiple est plus difficile, car la relation de sous-typage n'est plus arborescente. Il faut une condition supplémentaire pour garantir l'existence d'une définition la plus précise.

Slide 17

Un message est de la forme $\{m_i : a_i \to b_i\}_{i \in I}$ où a_i est un tuple de types (le type des arguments passés simultanément).

Il faut s'assurer que l'ensemble $\{a_i, i \in I \mid a < a_i\}$ est vide ou admet un plus petit élément pour tout type a. Formellement,

$$\forall a, \forall i, j \in I, a < a_i \wedge a < a_j \Longrightarrow \exists k \in I, a < a_k \wedge a_k < a_i \wedge a_k < a_j$$

Pour cela, il suffit que pour toute paire a_i, a_j de types incompatibles (de l'ensemble des domaines de m), pour tout type maximal a de l'ensemble de leurs sous-types communs, il existe une implémentation m_k de m de domaine a.

Contre-exemple

Slide 18

On considère les opérations binaires définies sur les entiers ("int") et les flottants ("float") avec la relation `int` < `float`. Le message `plus` est définit par les deux méthodes :

```
method plus (x@float, y@int) = x +. float_of_int y
method plus (x@int, y@float) = float_of_int x +. y
```

Ce message est mal formé. En effet, il est défini pour les types `int` × `float` et `float` × `int` dont le seul sous-type commun est `int` × `int` qui est donc maximal, mais pour lequel il n'y a pas d'implémentation.

Il faut donc compléter le message `plus` par une définition pour un argument de type `int` × "*int*".

En effet, si le message est appliqué à un argument de ce type, quelle branche faudrait il choisir ?

Cohérence

Slide 19

Pour assurer la cohérence entre le typage statique et l'exécution du programme, il faut imposer une contrainte sur la façon dont les messages peuvent être formés.

Dans l'application d'un message $\{m_i\}_{i \in I}$, l'analyse statique choisit la meilleure branche $m_1 : a_1 \to b_1$ compte tenu de l'information statique connue sur l'argument. Si le type de l'argument à l'exécution est plus précis, une branche plus spécifique $m : a_2 \to b_2$ peut être sélectionnée. Cela est possible si $a_2 < a_1$. Le résultat à l'exécution sera de type b_2, au lieu du type attendu b_1.

La cohérence impose que pour chaque paires de méthodes $m_i : a_i \to b_i$ et $m_i : a_j \to b_j$ composants un message m, si $a_i < b_i$ alors $b_i < b_j$.

Co-variance et contra-variance

Slide 20

Il faut distinguer
- Le recouvrement d'une méthode par une autre (spécialisation) qui est covariant.
- Le remplacement d'une méthode par une autre qui est contra-variant.

Dans le premier cas, la méthode ajoutée ne couvre pas tous les cas de la méthode qu'elle recouvre, l'ancienne méthode reste disponible pour les cas non couverts par la nouvelle, et il faut donc simplement une cohérence entre le résultat des deux méthodes.

Dans le second cas, la première méthode n'est plus disponible, d'où la contra-variance, puisqu'il faut que le domaine de la méthode de remplacement soit plus grand que le domaine de la méthode originale.

Contre-exemple à la cohérence

Slide 21

Imaginons que l'on complète la définition précédente par

```
method plus (x@int, y@int) = num_of_int x +/ num_of_int y
```

où `num` est le type des nombres en précision arbitraire, et supposons que `int` $<$ `num`.

Cette définition viole l'hypothèse de cohérence. En effet, le message "plus" comporte deux branches de domaines `int` $\times$ `int` et `int` $\times$ `float` en relation de sous-typage, mais leur codomaine respectif, "num" et `float` ne sont pas en relation de sous-typage (à l'inverse `float` $<$ `num`).

On pourrait alors appliquer le message à une valeur de type dynamique `int` $\times$ `int` mais de type statique `int` $\times$ `float`. On s'attend statiquement à récupérer une valeur de type `float`, mais on reçoit dynamiquement une valeur de type `num`.

Coercions

Slide 22

La coercion telle que `p@point` n'est plus un simple cast (changement du type statique, qui n'aurait pas d'effet sur le comportement dynamique).

Il doit changer le type dynamique de l'objet pour que la résolution se fasse dans comme si l'objet appartenait à une classe parente.

Bien sûr elle ne peut se faire que si le type statique de l'objet `p` est un sous-type du type vers lequel il est coercé (ce qui implique que la représentation de `p` est compatible (a plus de champs, et les valeurs des champs communs sont compatibles) avec la représentation du type vers lequel il est coercé.

Encapsulation

Slide 23

Pour retrouver une forme d'encapsulation, on peut replacer les méthodes dans les classes. Les méthodes ont alors accès aux variables d'état internes qui peuvent être cachées à l'extérieur qui ne voit l'objet que comme un type abstrait.

Cette approche peut être mixte. Les méthodes qui n'accèdent pas à l'état interne peuvent être placées en dehors de la classe, ce qui donne la possibilité d'une extension transparente (sans avoir à créer de sous-classe).

Mais cette solution reste imparfaite. L'encapsulation peut aussi être ignorée au niveau des objets et laissé à un mécanisme de modules.

Quelques exemples dansle langage Nice

Slide 24

Le langage Nice est un prototype de langage objets avec méthodes comme fonctions surchargées et multi-méthodes. Il se présente comme une extension de Java, avec un système de typage très riche, incorporant du polymorphisme paramétrique.

Ce n'est encore qu'un prototype...

On pourra voir le tout petit tutorial en ligne et quelques exemples.

Sur les machines de la salle 31 (noms d'oiseau : autruche, epervier, mouette, etc.)

C'est installé comme un package Java 1.2 qui se trouve dans `~remy/nice`. Le compilateur s'appelle `~remy/nice/bin/nicec` et l'interprétre de byte code `~remy/nice/bin/nicer`.

Nice : premier programme

Slide 25

On ne peut compiler que des packages

hello/main.nice

```
package hello;
main (args) {
   System.println ("hello, world!");
}
```

hello/main.nice

Unix

```
~remy/nice/bin/nicec hello
~remy/nice/bin/nicer hello.jar
```

Unix

Nice : les points

Une classe ne déclare que les types des variables d'instances

point/point.nice

```
class Point {
  int x;
}
```

point/point.nice

Slide 26

Un constructeur de la classe est défini comme une fonction de construction. On précise d'abord son type, puis son implémentation (en dehors de toute classe)

```
Point point (int);
point (x0) {
  Point p = new Point();
  p.x = x0;
  return p;
}
```

Nice : les méthodes

De même on définit les méthodes en dehors de la classe, en donnant d'abord son type, puis leur implémentation.

```
int getx (Point);
getx (p @ Point) { return p.x; }

void setx (Point, int);
setx (p @ Point, y) { p.x = y; }
```

Slide 27

On peut même surcharger la fonction `print` de la librairie `nice.lang` :

```
print (p @ Point) { System.out.print (getx (p)); }
```

On aurait pu aussi choisir un autre nom, bien sûr ; dans ce cas, il aurait fallu déclarer le prototype.

Nice : point coloré

Slide 28

```
class Cpoint extends Point { String c; }
Cpoint cpoint (String);
cpoint (s0) {
  Cpoint p = new Cpoint();
  p.x = 0;
  p.c = s0;
  return p;
}
String getc (Cpoint);
getc (cp @ Cpoint) { return cp.c; }

print (cp @ Cpoint) {
  System.out.print (getx(cp)); System.out.print (getc(cp));
}
```

Nice : l'égalité sur les points

Slide 29

On définit simplement une méthode à deux arguments :

```
boolean equals (Point, Point);
equals (p @ Point, q @ Point) {
  return p.x == q.x;
}
equals (p @ Cpoint, q @ Cpoint) {
  return p.x == q.x && p.c == q.c;
}
```

Nice : le test

points/main.nice

```
package point;

main (args) {
  System.out.print ("p = ");
  Point p = point (1);
  setx (p, getx(p)+1);
  print(p);
  System.out.print ("\ncp = ");
  Cpoint cp = cpoint ("black");
  print(cp);
  System.out.println ("\np = cp ? "
                    + (equals (p, cp) ? "yes" : "no"));
}
```

points/main.nice

Slide 30

Nice : autres exemples

Le système de typage contient des type paramétrés.

Voir par exemple la classe des collections.

Exercice 30 (Nice) *Essayer le langage nice. Mettre en place l'exemple des points colorés.*

En quoi l'implémentation de l'égalité en `Nice` *est-elle préférable à celle en Java ? en Ocaml ?* Réponse
Le modifier pour faire des points à deux dimensions.

Regarder l'exemple des Collections. □

Exercice 31 (Extensibilité bidirectionnelle) *On reprend l'exercice sur les listes en Nice. En particulier, on pourra définir les classes* `Nil` *et* `Cons`*, puis certaines méthodes sur ces classes, puis définir la classe* `Append`*, puis une nouvelle méthode sur l'ensemble des trois classes.* □

Slide 31

Résumé (inconvénients)

L'approche à surcharge est, malgré ses apparences, beaucoup plus difficile, par le caractère global et plus implicite de la récursion et par son mécanisme de résolution.

Elle rend également l'encapsulation plus difficile à moins de replacer les méthodes dans les classes.

Slide 32 La compilation séparée et le typage incrémentale sont plus difficiles et rarement implémentés.

Sa compilation efficace est aussi plus difficile en raison de la résolution dynamique, mais ce problème, qui reste secondaire et souvent résolu par une compilation globale.

Résumé (avantages)

L'approche à surcharge est plus proche de la programmation traditionnelle où les fonctions sont définies séparément des données et s'appliquent à celles-ci.

Elle offre plus de flexibilité dans l'écriture du code source puisque les méthodes peuvent être définies en dehors des classes.

Slide 33 Les multi-méthodes préservent la symétrie des arguments, contrairement aux méthodes binaires dans l'approche à enregistrement.

Conclusions

Slide 34

En conclusion, il faut souligner l'importance du typage et d'une étude formelle des constructions du langage.

Même en suivant cette approche (ce que nous avons fait qu'informellement ici) on rencontre de sérieuses difficultés.

Les objets dans le modèle à enregistrement sont déjà une notion difficile. Les messages comme fonctions surchargés le sont encore bien davantage, malgré, l'apparence trompeuse d'une intuition plus simple.

Enfin, la complexité du modèle trouvé ne doit pas faire oublier la simplicité du modèle recherché, mais celui-ci reste encore à trouver...

9

Représentation des objets (dans le modèle à enregistrement).

Cours	Exercices

Slide 1

Résumé

Les objets sont des valeurs sophistiquées. Ils ne sont pas représentables directement (en machine).

Comment faut-il les représenter en machine pour permettre une exécution efficace ?

Slide 2 Nous nous limitons au modèle à enregistrements. Nous considérons le cas général, et quelques cas particuliers intéressants.

Représentation des objets

Un objet est créé par instantiation d'une classe ou par clonage.

En théorie, il est représenté par une *copie* de l'environnement (enregistrement) représentant sa classe : l'affectation d'un champ d'un objet n'affectera pas les autres objets de sa classe.

Slide 3 En pratique, on partage la partie commune à chaque objet d'une même classe, i.e. l'ensemble des méthodes : un objet est alors un vecteur $(C, v_1, ..., v_k)$ où

– C est la représentation de la classe (*i.e.* un enregistrement anonyme) mais où les variables x_i ont été remplacées par leur position d_i dans le tuple, et

– v_{d_i} est la valeur de la variable x_i.

L'envoi de messages et l'accès aux variables d'instance doivent être modifiés de façon cohérente.

Appel de méthodes et accès aux variables

Slide 4

Les méthodes sont des fonctions anonymes placées dans les champs d'un enregistrement. C'est la base de la programmation par envoi de message : le comportement à effectuer à la réception d'un message, donc la fonction à exécuter est sélectionnée dynamiquement[a] dans l'enregistrement des méthodes porté par l'objet qui reçoit le message.

L'envoi d'un message $p\#m\ x_1\ \ldots\ x_n$ revient à appliquer la fonction représentant la méthode m dans p à l'objet p et aux arguments, soit $p.(0).m\ p\ x_1\ \ldots\ x_n$.

L'accès à une variable x_i est $p.(d_i)$, soit $p.(p.(0).x_i)$.

Le coup de l'envoi de message ou de l'accès à une variable est une ou deux indirections plus un accès dans un enregistrement anonyme.

[a] Sauf, dans certains cas où l'on peut déterminer cette fonction statiquement

Exemple

Slide 5

Code source

```
class point x =
  object (self)
    val mutable abscisse = x
    method getx = abscisse
    method bump = abscisse <- self#getx +1
  end
```

Traduction (non typée)

```
let getx_point self = self.(self.(0).abscisse)
let bump_point self =
  self.(0).abscisse <- self.(0).getx self + 1
let partagé =
  { abscisse = 1; point = getx_point; bump = bump_point }
let new_point x : obj = [| partagé; x |]
```

Exemple

Slide 6

Envoi de message

```
let p = new point 17 in p#bump
```

Traduction

```
let p = new_point 17 in p.(0).bump p
```

Se réduit en

```
let p = [| meth; 17 |] in bump_point p
```

puis

```
[| meth; 18 |]
```

Enregistrements anonymes

Slide 7

Les enregistrements utilisés ici sont anonymes, *i.e.* non déclarés. Comment les représenter ? Dans le cas général, on ne connaît pas la position associée à un champ de l'enregistrement, car on ne connaît que son type (qui est une information incomplète en présence de sous-typage).

L'utilisation des enregistrements anonymes est nécessaire pour que l'envoi de message soit polymorphe.

Dans le cas général, un enregistrement anonyme est une table

$$\ell \in etiquettes \mapsto v \in valeurs$$

La représentation d'une telle table peut être coûteuse. La représentation des enregistrements et des objets se ramène à
– choisir une représentation générale relativement efficace,
– optimiser les cas particuliers fréquents.

Cas général

En fait, il s'agit de fournir une représentation et des opérations efficaces pour un ensemble de p enregistrements (classes) avec au plus q étiquettes.

De façon abstraite, cela se ramène à la représentation d'une fonction partielle :

$$c \in [1, p] \times \ell \in [1, q] \rightharpoonup d \in I\!N$$

Slide 8

Cette table peut se décomposer dans l'une ou l'autre direction :

$$c \mapsto (\ell \rightharpoonup d), \qquad \text{ou} \qquad \ell \mapsto (c \rightharpoonup d).$$

La partie gauche peut toujours s'implanter par un vecteur.

Comment représenter la partie droite ?

Représentation par listes d'associations

C'est la solution la plus simple, incrémentale, efficace en espace, mais très coûteuse en temps de calcul surtout pour de gros enregistrements.

On peut l'améliorer en utilisant

1. des arbres balancés.
2. des caches (le dernier appel est remis en tête).

Slide 9

Avec utilisation de caches, la solution est relativement performante surtout en utilisant la décomposition $\ell \mapsto (c \rightharpoonup d)$.

Représentation par des vecteurs

Chaque méthode est un vecteur de taille voisine de q. Solution efficace en temps, mais trop coûteuse en espace.

Pour gagner en place, on peut superposer

1. Deux étiquettes qui ne sont jamais dans un même enregistrement.
2. Deux enregistrements qui ont des domaines disjoints.

Slide 10

La compaction 1 est efficace, mais pas incrémentale. La compaction 2 est incrémentale, mais peu efficace, car certaines méthodes comme `print` sont dans presque tous les enregistrements)

Compaction au prix d'une indirection

On peut rendre la compaction 2 très efficace au prix d'une indirection supplémentaire en coupant $\ell \in [1, q] \rightharpoonup d$ en

$$\ell_{fort} \in [1, q_1] \rightharpoonup (\ell_{faible} \in [1, q_2] \rightharpoonup d) \qquad \text{où } q_1, q_2 \approx \sqrt{q}$$

Beaucoup plus d'opportunités de compaction. Jamais de gros trous.

C'est la solution utilisée dans Ocaml.

Slide 11

Message d'une classe connue

Lorsque qu'on envoie le message à un objet d'une classe connue, on peut calculer statiquement l'emplacement de la méthode.

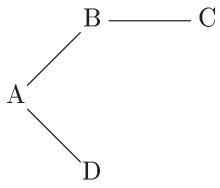
Ce cas est assez fréquent, surtout si on fait au préalable une analyse de flux.

Slide 12

Cas particulier de l'héritage simple

Une classe n'a qu'un seul *parent*. Cela permet de placer les champs à des positions réservées dans toutes les sous-classes.

Slide 13



A	a			
B	a	b_1	b_2	
C	a	b_1	b_2	c
D	a	b_1	b_2	d

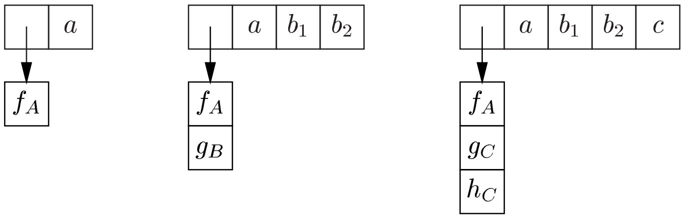
L'accès à une variable d'instance $p.x$ peut être optimisé lorsque la position de x est connue statiquement, c'est-à-dire dès que l'on connaît une classe parente de l'objet qui possède ce champ (1).

Dans l'exemple ci-dessus, un accès à la variable a dans un objet d'une sous-classe de B est uniformément $p.(1)$.

Accès aux méthodes

Cette optimisation s'étant aux méthodes (avec une indirection).

Slide 14



Un appel de méthode $p\#g\ v_1\ v_2$ dans un objet p dont on sait qu'il est une sous-classe de B est $p.(0).(1)\ p\ v_1\ v_2$.

Avantages et limitations

Slide 15

Les avantages

+ L'héritage simple concerne de nombreux langages (Java compris).
+ L'optimisation est très efficace.

Les limitations : il faut que la condition (1) soit satisfaite :

- En général, on utilise un système de typage qui restreint le sous-typage à du sous-classage. Ce qui n'est pas un très bon choix...
- Cela est incompatible avec la possibilité de cacher des variables d'instances.
- Cela ne fonctionne pas dans un langage non typé.

Test d'appartenance

Appartenance au sens strict Il s'agit de tester si un objet à été créé par la classe C. Pour cela, il suffit de mettre dans chaque objet une information indiquant sa classe. On peut utiliser le champ $p.(0)$ à cet effet.

Appartenance au sens large Il s'agit de tester si un objet p appartient au sens strict à une sous-classe de C.

Slide 16

Si on se restreint au cas de l'héritage simple, cela revient à tester si la classe C_0 de p est une sous-classe de C, ou de façon abstraite, à savoir si le nœud C_0 domine le nœud C dans l'arbre d'héritage.

Test d'appartenance (héritage simple)

Il suffit de rechercher si C est une des classes parentes de C_0. On peut effectuer ce test en temps constant en représentant les classes parentes de C_0 dans un vecteur V (une classe étant positionnée à sa profondeur dans la hiérarchie), et à condition de connaître la profondeur d de la classe C. Alors, C_0 est sous-classe de C si et seulement si $V.(d) = C$.

Slide 17

Langages sans classes

Il existe des langages sans classes, mais mis à part les langages théoriques, ils sont plus rares. Les objets sont créés par extension ou duplication des modèles. Ils peuvent se compiler par la méthode générale.

La création d'un nouveau type d'objet revient à la création d'une nouvelle classe.

Slide 18

D Solution des exercices

Exercice 1, page 12

Les types des monnaies Euro.t et Dollar.t sont abstraits donc incompatibles. Il n'est donc plus possibles d'écrire une fonction de coercion.

La solution est de donner dans un premier temps des versions concrète des modules Euro et Dollar, implémenter les fonctions de coercion puis d'exporter les modules Euro et Dollar de façon abstraite.

Exercice 1 (continued)

```
module type BUREAU_DE_CHANGE =
  sig
    module Euro : MONNAIE
    module Dollar : MONNAIE
    val euro_to_dollar : Euro.t -> Dollar.t
    val dollar_to_euro : Dollar.t -> Euro.t
  end;;
```

Exercice 1 (continued)

```
module Bureau_de_change : BUREAU_DE_CHANGE =
  struct
    module Euro = Float
    module Dollar = Float
    let un_euro_en_dollar = 0.95
    let euro_to_dollar x = x /. un_euro_en_dollar
    let dollar_to_euro x = x *. un_euro_en_dollar
  end;;
```

Exercice 1 (continued)

```
open Bureau_de_change;;
let deux_euro = Euro.prod 2.0 Euro.un;;
let deux_euro_en_dollar = euro_to_dollar deux_euro;;
let _ = euro_to_dollar Dollar.un;;
```

Exercice 2, page 27

amis.mlUne solution

Exercice 3, page 28

```
type 'a pile = 'a list ref;;
let créer() = ref [];;
let ajouter x p = p := x :: !p;;
exception Vide;;
let retirer p =
  match !p with
  | [] -> raise Vide
  | x::t -> p:= t; x;;
```

Exercice 3 (continued)

Il suffit de placer le code précédent dans un fichier `pile.ml` et de définir l'interface suivante dans une fichier `pile.mli`.

pile.mli

```
type 'a pile
val créer : unit -> 'a pile
exception Vide
val ajouter : 'a -> 'a pile -> unit
val retirer : 'a pile -> 'a
```

pile.mli

Exercice 3 (continued)

On peut définit la consultation comme le retrait d'un élément suivi de son ajout ; les autres composantes sont inchangées :

cpile.ml

```
type 'a pile = 'a Pile.pile
let créer = Pile.créer
(* la ligne suivante, necessaire, n'est pas possible dans la version 2.04 *)

exception Vide = Pile.Vide
let ajouter = Pile.ajouter
let retirer = Pile.retirer
let consulter p = let x = Pile.retirer p in Pile.ajouter x p; x;;
```

cpile.ml

cpile.mli

```
type 'a pile
val créer : unit -> 'a pile
exception Vide
val ajouter : 'a -> 'a pile -> unit
val retirer : 'a pile -> 'a
val consulter : 'a pile -> 'a
```

cpile.mli

Note : Dans les versions antérieure à 3.00, le renommage d'une exception "`exception Vide = Pile.Vide`" n'était pas possible. Le seul moyen de contourner le problème est alors d'exporter une fonction permettant d'attraper une exception (un *handler*) :

```
let attrape_vide f x f' x' =
  try f x with Vide -> f' x';;
```

dans le fichier `pile.ml` avec l'interface suivante :

```
val attrape_vide : ('a -> 'b) -> 'a -> ('c -> 'b) -> 'c -> 'b
```

Et de ré-exporte `attrape_vide` dans `cpile.ml`.

Exercice 3 (continued)

Pour écrire une version plus efficace, il faudrait connaître la réprésentation des piles.

piles.ml

```
module Pre_Pile =
  struct
    type 'a pile = 'a list ref
    let créer() = ref []
    let ajouter x p = p := x :: !p
    exception Vide
    let retirer p =
      match !p with
```

```
      | [] -> raise Vide
      | x::t -> p:= t; x
  end;;

module type PILE =
  sig
    type 'a pile
    val créer : unit -> 'a pile
    exception Vide
    val ajouter : 'a -> 'a pile -> unit
    val retirer : 'a pile -> 'a
  end;;

module Pile = (Pre_Pile : PILE);;

module type CPILE =
  sig
    include PILE
    val consulter : 'a pile -> 'a
  end

module Cpile : CPILE = struct
  open Pre_Pile
  type 'a pile = 'a Pre_Pile.pile
  let créer = créer
  exception Vide = Vide
  let ajouter = ajouter
  let retirer = retirer
  let consulter p =
    match !p with
    | [] -> raise Vide
    | x::t -> x
end;;
```

piles.ml

Exercice 4, page 30

polynome.mli

```
(* la structure d'anneau *)
module type ANNEAU =
  sig
    type t                    (* type des éléments de l'anneau *)
    val zéro : t
    val unité : t
    val nég : t -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
  end
      ;;

(* la structure d'anneau sur des valeurs de type t *)
```

```
module type POLYNOME =
  sig
    type c
        (* type des coefficients *)
    type t
        (* type des polynômes *)
    val zéro : t
    val unité : t
        (* unité pour le produit des polymôme.
           Superflue, car c'est un cas particulier de monôme, mais cela
           permet à la structure de polynôme d'être une superstructure de
           celles des anneaux *)
    val nég : t -> t
    val monôme : c -> int -> t
        (* monôme a k retourne le monôme a X^k *)
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
    val eval : t -> c -> c
  end
      ;;


(* Le fonction qui étant donné une structure d'anneau retourne une structure
de polynôme. Il faut faire attention à réexporter le types des coefficients
de façon partiellement abstraite afin que les opérations sur les coefficients
des polynômes puissent être manipulés de l'extérieur *)


module Make (A : ANNEAU) (X : sig val nom_variable : string end)
    : (POLYNOME with type c = A.t)
    ;;
```

polynome.mli

polynome.ml

```
(* la structure d'anneau *)
module type ANNEAU =
  sig
    type t
    val zéro : t
    val unité : t
    val nég : t -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
  end
      ;;

(* la structure d'anneau sur des valeurs de type t *)
module type POLYNOME =
```

```
  sig
    type c
    type t
    val zéro : t
    val unité : t
    val nég : t -> t
    val monôme : c -> int -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
    val eval : t -> c -> c
  end
      ;;

module Make (A : ANNEAU) (X : sig val nom_variable : string end) =
  struct
    type c = A.t

    (* un monome est un coeficient et une puissance *)
    type monôme = (c * int)
    (* un polynome est une liste de monomes triés par rapport au puissance *)

    (* il est essentiel de préserver cet invaraint dans le code ci-dessous *)
    type t = monôme list

    (* pour que la représentation soit canonique, on élimine aussi
       les coeficients nuls, en particulier le monome nul est la liste vide *)
    let zéro = []

    (* on peut donc (grâce à l'invariant utiliser l'égalité générique *)
    let rec equal p1 p2 =
      match p1, p2 with
      | [],[] -> true
      | (a1, k1)::q1, (a2, k2)::q2 -> k1 = k2 && A.equal a1 a2 && equal q1 q2
      | _ -> false

    (* création d'un monôme *)
    let monôme a k =
      if k < 0 then failwith "monôme: exposant négatif"
      else if A.equal a A.zéro then [] else [a, k]

    (* un cas particulier l'unité *)
    let unité = [A.unité, 0]

    (* attention à respecter l'invariant et ordonner les monômes *)
    let rec plus u v =
      match u, v with
        (x1, k1)::r1 as p1, ((x2, k2)::r2 as p2) ->
          if k1 < k2 then
            (x1, k1):: (plus r1 p2)
          else if k1 = k2 then
            let x = A.plus x1 x2 in
```

```
        if A.equal x A.zéro then plus r1 r2
        else (A.plus x1 x2, k1):: (plus r1 r2)
      else
        (x2, k2):: (plus p1 r2)
  | [], _ -> v
  | _ , [] -> u

(* on pourrait faire beaucoup mieux pour éviter de recalculer les
   puissances de $k$, soit directement, soit en utilisant une
   mémo-fonction *)

let rec fois (a, k) = (* on suppose a <> zéro *)
  function
    | [] -> []
    | (a1, k1)::q ->
        let a2 = A.mult a a1 in
        if A.equal a2 A.zéro then fois (a,k) q
        else (a2, k + k1) :: fois (a,k) q

(* on pourrai écrire : ``fois (A.nég A.unité) p'' *)
let nég p = List.map (function a, k -> A.nég a, k) p

let mult p = List.fold_left (fun r m -> plus r (fois m p)) zéro

(* une impression grossière *)
let string_monome (c,k) =
  if k = 0 then A.string c
  else if A.equal A.zéro c then "0"
  else
    let string_k =
      if k = 1 then X.nom_variable
      else String.concat "^" [ X.nom_variable; string_of_int k] in
    if A.equal A.unité c then string_k else (A.string c)^string_k
let string p =
  if p = [] then "0"
  else "(" ^ (String.concat " + " (List.map string_monome p)) ^ ")"

(* Puissance c^k, c un coefficient, k un entier >= 0. Par dichotomie. *)
let rec puis c = function
  | 0 -> A.unité
  | 1 -> c
  | k ->
      let l = puis c (k lsr 1) in
      let l2 = A.mult l l in
        if k land 1 = 0 then l2 else A.mult c l2

let eval p c = match List.rev p with
| [] -> A.zéro
| (h::t) ->
    let (* Réduire deux monômes en un. NB: on a k >= l. *)
        dmeu (a, k) (b, l) =
      A.plus (A.mult (puis c (k-l)) a) b, l
    in
```

```
      let a, k = List.fold_left dmeu h t in
      A.mult (puis c k) a

  end
   ;;
```

polynome.ml

Exercice 4 (continued)

polytest.ml

```
(* pour charger le fichier compilé polynome.cmo dans le mode interactif *)

#cd "../3";;
#load "polynome.cmo";;
open Polynome;;

(* l'anneau des entiers *)
module I =
  struct
    type t = int
    let zéro = 0
    let unité = 1
    let nég x = - x
    let plus = ( + )
    let mult = ( * )
    let equal = ( = )
    let string = string_of_int
end;;

(* l'anneau des Z/2Z *)
(* invariant: on maintient la représentation canonique 0 ou 1 *)
module Z_2Z =
  struct
    type t = int
    let zéro = 0
    let unité = 1
    let plus x y = (x + y) mod 2
    let nég x = x
    let mult x y = (x * y) mod 2
    let equal = ( = )
    let string = string_of_int
end;;

(* On abstrait le test pour le répéter avec différents anneaux de base *)

module Test(A: ANNEAU) =
  struct
    (* les polynômes à coefficients dans A *)
    module X = Make (A)(struct let nom_variable = "X" end)

    (* A toplevel, poour mettre au point:
```

```
    let print_X x = Format.print_string (X.string x)
    #install_printer print_X
  *)

  (* on construit (1 + X)(1 - X) = 1 - X^2 *)
  let ( ^ ) = X.monôme
  let ( + ) = X.plus
  let ( * ) = X.mult
  let ( - ) p q = p + (X.nég q)
  let z = A.unité
  let x0, x1, x2 = z^0, z^1, z^2
  let p = (x0 + x1) * (x0 - x1)
  let q = x0 - x2
  let () =
    Printf.printf "%s %s %s\n"
      (X.string p) (if X.equal p q then "=" else "<>") (X.string q)

 (* les polynomes à coefficients dans X *)
  module Y = Make (X)(struct let nom_variable = "Y" end)

 (* Idem
    let print x = Format.print_string (Y.string x)
    #install_printer print *)

  let ( ^^ ) = Y.monôme
  let ( ++ ) = Y.plus
  let ( ** ) = Y.mult
  let ( -- ) p q = p ++ (Y.nég q)
  let x1, x2, y1, y2 = x1^^0, x2^^0, x0^^1, x0^^2
  let p = (x1 ++ y1) ** (x1 -- y1)
  let q = x2 -- y2

  let () =
    Printf.printf "%s %s %s\n"
      (Y.string p) (if Y.equal p q then "=" else "<>") (Y.string q)

end;;


(* Le test pour I et pour Z_2Z *)
let () = print_string "\n(* Dans l'anneau des entier*)\n\n" in
let module T1 = Test(I) in
let () = print_string "\n(* Dans l'anneau Z/2Z *)\n\n" in
let module T2 = Test(Z_2Z) in
();;
```

polytest.ml

Exercice 4 (continued)

poly.ml

```
open Polynome

(* L'anneau des flottants --c'est plus intéressant que les entiers... *)
module F =
```

```
  struct
    type t = float
    let zéro = 0.
    let unité = 1.
    let nég x = -. x
    let plus = ( +. )
    let mult = ( *. )
    let equal = ( = )
    let string = string_of_float
  end;;
module X = Make (F) (struct let nom_variable = "X" end);;

(* construction d'un polynome: le coefficients en X^k est représenté
   par un flottant à l'indice k du tableau *)
let poly t =
  Array.fold_left
    (fun r (s, k) -> X.plus (X.monôme (float_of_string s) k) r)
    X.zéro
    (Array.mapi (fun i x -> (x, i)) t)
    ;;

(* il faut enlever le premier argument qui est le nom de la commande *)
let suffix t k = Array.sub t k (Array.length t - k);;

let main () =
  let écho, k =
    if Array.length Sys.argv > 1 && Sys.argv.(1) = "-v" then true, 2
    else false, 1 in
  let t = suffix Sys.argv k in
  let p = poly t in
  let print_eval x =
    if écho then Printf.printf "P(%f) = " x;
    print_float (X.eval p x); print_newline() in
  if écho then Printf.printf "P = %s\n" (X.string p);
  while true do print_eval (float_of_string (read_line())) done
    ;;

try main() with End_of_file -> ();;
```

poly.ml

poly.sh

```
#!/bin/sh
#
#   Exécuter les commandes sous le shell ou sauver ce fichier
#   dans poly.sh et faire sh poly.sh

./poly 1. 2. 1. <<EOF
0.5
1.
1.5
2.
2.5.
3.
```

```
EOF
```
poly.sh

Exercice 4 (continued)

Voici une solution avec le Makefile générique.

Exercice 5, page 8

Il n'y a pas d'intérêt a priori, sauf situation très particulière. Cette transformation vient surtout des langages tout objets qui n'ont pas de fonctions mais seuleument des méthodes. En Ocaml, les fonctions sont faciles à définir et à manipiler. Lorsque cette approche suffit, elle est plus légère que le style à objets et donc préférable.

Exercice 5 (continued)

Lorsque des fonctions polymorphes sont mises dans une classes ont voudrait en faire des méthodes polymorphes. Mais ce n'est pas possibles car les objets ne sont jamais polymorphes (comme des références, ils contiennent des champs mutables), et leurs méthodes non plus.

Au mieux on pourrait définir une classe paramétrique (par exemple des opérations sur les listes), mais ils faudra spécialiser celles-ci complétement à la création d'un objet. C'est une limitation forte à l'usage des classes pour regrouper des fonctions de librairie.

Exercice 6, page 30

On ne donne que la solution pour des classes paramétriques représentées par des listes.

```
exception Vide;;
class ['a] pile =
  object
    val mutable p : 'a list = []
    method ajouter x = p <- x :: p
    method retirer =
      match p with
      | [] -> raise Vide
      | x::t -> p <- t; x
  end;;
```

Exercice 7, page 39

Les noms de méthodes cette autre classe ne doivent pas être en conflit avec ceux de la classe sauvegarde.

Exercice 7 (continued)

Il suffit lorsqu'on prend une copie de mettre le champ copie à `None` :

```
class une_sauvegarde =
    object (self : 'mytype)
      inherit sauvegarde
      method sauve = copie <- Some {< copie = None >}
    end;;
```

Si une seule copie est nécessaire, la version ci-dessus libère les anciennces versions qui peuvent être récupérées par le GC. Dans la version précédentes toutes les versions intermédiaires restaient vivantes aussi longtemps que la version de travail.

Exercice 7 (continued)

```
class fsauvegarde =
  object (self)
    val original = None
    method copie = {< original = Some self >}
    method récupère =
      match original with None -> self | Some x -> x
  end;;
```

Exercice 7 (continued)

La méthode `balai` efface récursivement et physiquement certaines copie en les faisant pointer vers la copie précedente.

```
class sauvegarde_autogérée =
  object (self : 'mytype)
    inherit sauvegarde
    method private coup_de_balai p q =
      try
        let q' = if p = q then 2 * q else q in
        let la_sauvegarde = self # récupère # le_balai (succ p) q' in
        copie <- la_sauvegarde;
        if p = q then Some self else la_sauvegarde
      with Not_found -> None
    method balai = ignore (self#coup_de_balai 1 1)
  end;;
```

Remarquer l'importance de faire une méthode auxilliaire de `le_balai`. Une fonction ne conviendrait pas, car `le_balai` utilise la variable d'instance copie.

Exercice 8, page 43

La méthode concat pose problème parce que c'est une méthode binaire. En fait, c'est une pseudo méthode binaire. En effet, si on naturellement :

```
class ostring s = object (self)
  val s = s
  method repr = s
  method concat t = {< s = s ^ t # repr >}
end;;
```

alors l'argument `t` n'est pas nécessairement du type de self, mais doit seulement posséder une méthode repr du bon type. Cela rend la class paramétrique (d'où le message d'erreur). Une solution est donc :

```
class ['a] ostring s = object (self)
  val s = s
  method repr = s
  method concat (t:'a) = {< s = s ^ t # repr >}
end;;
```

Une autre solution, sans doute meilleurs en pratique, et de contraindre le type 'a de t a être le même que le type de self (la classe n'est plus paramétrique)

```
class ostring s = object (self : 'a)
  val s = s
  method repr = s
  method concat (t:'a) = {< s = s ^ t # repr >}
end;;
```

Attention ! on pourrait aussi contraindre `t` à être du type de la classe `string`

```
class ostring s = object (self : 'a)
```

```
  val s = s
  method repr = s
  method concat (t : ostring) = {< s = s ^ t # repr >}
end;;
```
Cette solution n'est pas bonne car dans une sous-classe enrichie `ostring_plus`, la méthode `concat` ne se comportera pas comme attendu : elle prendra et un argument du type `ostring` et non pas du type `ostring_plus`.

Exercice 9, page 44

Le problème éventuel est le conflit de nom, si par exemple, la classe `euro_compte` utiliser les mêmes méthodes `dépôt` et `retrait` que la classe `compte`.

Malheureusement, il n'est pas possible (pour l'instant, en Ocaml) de renommer les méthodes.

Le choix des noms est important en style objet, car la programmation par "envoi de messages" attache un sémantique implicite à chaque message et les noms sont bien plus que des conventions syntaxiques. Par exemple, un object imprimable aura naturellement une méthode "print", sinon il ne pourra pas être imprimé par d'autres classes définies en librairie.

La solution la plus simple consiste à prévoir ce conflit et choisir des noms différents pour chacune des méthodes.
```
class euro_compte = object
  val sole_en_euro = ...
  method dépôt_en_euro = ...
end
```
Une autre solution, plus modulaire, sinon plus élégante, permet toutefois de partager le compte en euro et en francs. Mais il faut pour cela prévoir le conflit, et utiliser un motif en peigne (cf introduction) avec une classe génératrice, où les noms sont privés, et des classes dérivées où les noms sont figés. (On choisi une version simplifié, mais on imaginera une version raffinée de la classe ou le corps de la méthode `dépôt` est suffisamment gros pour justifier le partage).
```
class compte = object
  val mutable solde = 0.
  method private dépôt x = solde <- solde +. x; solde
end;;
```
Pour la classe en franc, on choisit des nom précis
```
class type compte_français = object
  method dépôt_en_franc : float -> float
end

class compte_en_franc : compte_français = object (self)
  inherit compte
  method dépôt_en_franc = self#dépôt
end;;
```
Pour de façon similaire en évitant les conflits de noms :
```
class type compte_européen = object
  method dépôt_en_euro : float -> float
end

class compte_en_euro : compte_européen = object (self)
  inherit compte
  method dépôt_en_euro = self#dépôt
end;;
```
On peut alors mélanger les deux comptes :
```
class compte_mixte = object
  inherit compte_en_franc
```

```
  inherit compte_en_euro
end;;
```
On vérifie que la class `compte_mixte` comporte bien deux variables d'instances soldes cachées distinctes : il n'y a pas d'override entre les champs (variables ou méthodes) cachés. (Dans cette version simplifié du compte, on effectue un dépôt vide pour lire la valeur du compte) :
```
let cm = new compte_mixte;;
let soldes() = (cm # dépôt_en_franc 0., cm # dépôt_en_euro 0.);;
soldes();;
```
− : float ∗ float = 3.000000, 4.000000
```
cm # dépôt_en_franc 1.;;
```
− : float = 4.000000
```
soldes();;
```
− : float ∗ float = 4.000000, 4.000000
```
cm # dépôt_en_euro 2.;;
```
− : float = 6.000000
```
soldes();;
```
− : float ∗ float = 4.000000, 6.000000

NOTE : On a ainsi résolu le problème de façon modulaire (partage du code), mais au pris d'un motif en peigne, qu'il a fallu concevoir a priori.

$$\begin{tikzcd}
G_1 \arrow[r, Rightarrow] \arrow[d, dash] & G_2 \arrow[r, Rightarrow] \arrow[d, dash] & G_3 \arrow[r, Rightarrow] \arrow[d, dash] & G_4 \arrow[d, dash] \\
C_1 \arrow[d, dotted, dash] & C_2 \arrow[d, dotted, dash] & C_3 \arrow[d, dotted, dash] & C_4 \arrow[d, dotted, dash] \\
O_1 & O_2 & O_3 & O_4
\end{tikzcd}$$

Exercice 10, page 10

Il suffit de faire une définition de type de classe. Dans les classes, les "nil" et "cons" les contraintes de types peuvent être simplifiées.
```
module Liste = struct
  exception Nil
  class virtual ['a] liste =
    object (_:'alist)
      method virtual hd : 'a
      method virtual tl : 'alist
      method virtual null : bool
      method virtual length : int
    end
  class ['a] nil =
    object
      inherit ['a] liste
      method hd = raise Nil method tl = raise Nil
      method null = true method length = 0
    end
  class ['a] cons h t =
    object
      inherit ['a] liste
      val hd = h val tl = t
      method hd = h
```

```
      method tl = t
      method null = false
      method length = 1 + tl#length
    end
end;;
```

Exercice 10 (continued)

On ajoute une cellule dans laquelle on mémorise les branches gauches et droites.

```
module L =
  struct
    class ['a] nil = ['a] Liste1.nil
    class ['a] cons = ['a] Liste1.cons
    class ['a] append l r =
      object (self : 'mytype)
        val left = (l : 'mytype)
        val right = (r : 'mytype)
        method null = left # null && right # null
        method hd =
          if left # null then right # hd
          else (left # hd : 'a)
        method tl =
          if left # null then right # tl
          else if left # tl # null then right
          else {< left = left # tl >}
        method length = left # length + right # length
      end
  end;;
```

La difficulté est la méthode `tl` de la classe `append`. Elle utilise la copie fonctionnelle `{< left = left # tl >}` qui crée un nouvel objet identique à `self` sauf dans lequel la branche gauche ("left") est remplacée par sa queue (`left # tl`).

Un petit test (on utilise les infixes pour que la notation soit plus lisible)

```
let ($) = new L.cons and (@) = new L.append in
  let l = (1 $ (2 $ new L.nil)) @ (3 $ (4 $ new L.nil)) in
  l # length;;
```

− : int = 4

Exercice 10 (continued)

```
module Cellule = struct
  exception Nil
  class ['a,'b] nulle =
    object
      method hd : 'a = raise Nil method tl : 'b = raise Nil
      method null = true
      method length = 0
    end
  class ['a,'b] cellule h t =
    object
      val hd : 'a = h val tl : 'b = t
      method hd = h method tl = t
      method null = false
      method length = 1 + tl#length
    end
```

```
end;;
module Liste3 = struct
  class ['a] nil =
    object (_:'b) inherit ['a, 'b] Cellule.nulle end
  class ['a] cons x y =
    object (_:'b) inherit ['a, 'b] Cellule.cellule x y end
end;;
```

Exercice 11, page 20

On récupère le code du compte avec relevé :

```
class compte =
  object
    val mutable solde = 0.0
    method solde = solde
    method dépôt x = solde <- solde +. x
    method retrait x =
      if x <= solde then (solde <- solde -. x; x) else 0.0
  end;;

class compte_avec_intérêts =
  object (self)
    inherit compte
    method private intérêts =
      self # dépôt (0.03 *. self # solde)
  end;;

class bon_compte =
  object
    inherit compte_avec_intérêts as super
    method dépôt x =
      if x > 0.0 then super # dépôt x
      else raise (Invalid_argument "dépôt")
  end;;

type opération = Dépôt of float | Retrait of float

class compte_avec_relevé =
  object (self)
    inherit bon_compte as super
    val mutable registre = []
    method private trace x = registre <- x::registre
    method dépôt x =
      self#trace (Dépôt x); super # dépôt x
    method retrait x =
      self#trace (Retrait x); super # retrait x
    method relevé = List.rev registre
  end;;
```

On définit maintenant le module Compte :

```
let date()= (1998,10,30);;

module Compte = struct
  class compte_bancaire = compte_avec_relevé
```

```
  class type vue_du_client = object
    method dépôt : float -> unit
    method relevé : opération list
    method retrait : float -> float
    method solde : float
  end
  class client x : vue_du_client =
    let y = if x >= 1000. then x
    else raise (Failure "Apport insuffisant") in
    object (self)
      inherit compte_bancaire initializer self#dépôt y end
  let promotion x =
    let c = new client x in
    if date() < (1998,10,30) then c # dépôt 100.; c
end;;
```

Exercice 11 (continued)

```
exception Découvert
class compte_géré ?seuil:(s=10000.) ?init:(i=0.) c =
  object (self)
    inherit Compte.client i as banque
    val épargne = c
    method investir x = épargne # dépôt (self # retrait x)
    method dépôt x =
      banque # dépôt x; if self#solde > s then
      self#investir (float (truncate s / 2))
    method retrait x =  (* pour ne pas avoir de pénalité *)
      if self#solde < x then raise Découvert else banque#retrait x
  end;;

let caisse_épargne = new Compte.client 2000.0;;
let ccp = new compte_géré ~init:1000. caisse_épargne;;
```

Exercice 12, page 10

```
class type p_sans_m =
  object method n : int end;;
class p : p_sans_m =
  object (self) method m = 1 method n = 1 + self#m end;;
```

Characters 23−75:
The class type object method m : int method n : int end
is not matched by the class type p_sans_m
The public method m cannot be hidden

```
class q = object inherit p method m = true end;;
```

Exercice 13, page 25

La méthode `bin` est définie avec les types suivants :

Type de x	A	B	
Type de y	A	A	B
Résultat	1	1	2

La seule combinaison possible qui retourne 2 est `b.bin(b)`. Sinon, l'objet `x` ou l'argument `y` est (éventuellement coercé en une valeur) de type `A` et le résultat retourné est `1`. Soit finalement

$x \setminus y$	aa	ab	bb
aa	1	1	1
ab	1	1	1
bb	1	1	2

Exercice 13 (continued)

```
class A          { int bin(A x){ return 1; } }
class B extends A { int bin(B x){ return 2; } }

class AB{
  static A aa = new A();
  static B bb = new B();
  static A ab = bb;
  static void print (String s, int r){
    System.out.print(s); System.out.println(r);
  }
  public static void main (String argv[]) {
    print ("aa.bin(aa)=",aa.bin(aa));
    print ("aa.bin(ab)=",aa.bin(ab));
    print ("aa.bin(bb)=",aa.bin(bb));
    print ("ab.bin(aa)=",ab.bin(aa));
    print ("ab.bin(ab)=",ab.bin(ab));
    print ("ab.bin(bb)=",ab.bin(bb));
    print ("bb.bin(aa)=",bb.bin(aa));
    print ("bb.bin(ab)=",bb.bin(ab));
    print ("bb.bin(bb)=",bb.bin(bb));
  }
}
```

Exercice 13 (continued)

La méthode première définition `bin(A x)` override (remplace) la définition de la classe `A` alors que la deuxième définition `bin(B x)` surcharge (complète) la définition précédente.

Exercice 13 (continued)

Lorsque le sujet (`x`) est de la classe `B` et que dans le premier cas, on la réponse n'était pas `2`. En effet, c'était c'est la méthode de la classe `A` qui était appelé, maintenant ce sera la méthode de `B` qui a le même type que dans classe `A` donc qui prend un argument de type `A`. Ainsi

$x \setminus y$	aa	ab	bb
aa	1	1	1
ab	3	3	3
bb	3	3	2

Exercice 13 (continued)

```
class A           { int bin(A x){ return 1; } }
class B extends A{
  int bin(A x){
    if (x instanceof B) return bin((B)(x));
```

```
    else return super.bin(x);
  }
  int bin(B x){ return 2; }
}

class AB{
  static A aa = new A();
  static B bb = new B();
  static A ab = bb;
  static void print (String s, int r){
    System.out.print(s); System.out.println(r);
  }
  public static void main (String argv[]) {
    print ("aa.bin(aa)=",aa.bin(aa));
    print ("aa.bin(ab)=",aa.bin(ab));
    print ("aa.bin(bb)=",aa.bin(bb));
    print ("ab.bin(aa)=",ab.bin(aa));
    print ("ab.bin(ab)=",ab.bin(ab));
    print ("ab.bin(bb)=",ab.bin(bb));
    print ("bb.bin(aa)=",bb.bin(aa));
    print ("bb.bin(ab)=",bb.bin(ab));
    print ("bb.bin(bb)=",bb.bin(bb));
  }
}
```

Le résultat est toujours 2 dès que `x` et `y` sont tous deux de la class B, indépendemment de leur type statique. En effet dès que `x` est de la classe B, la méthode `bin` est choisie dans la classe B indépendemment du type (statique) de `x`. Par ailleurs, les deux implémentations de `bin` dans la classe B retournent 2 dès que `y` est un objet de la classe B.

Exercice 13 (continued)

```
class A          { int bin_A (A x){ return 1; } }
class B extends A{
  int bin_A (A x){ return 3; }
  int bin_B (B x){ return 2; }
}
```

Exercice 13 (continued)

$x \setminus y$	aa	ab	bb
aa	bin_A	`bin_A`	bin_A
ab	bin'_A	bin'_A	bin'_A
bb	bin'_A	bin'_A	bin'_B

Exercice 16, page 52

```
class ['a] cell x =
  object
    val mutable contents : 'a = x
    method get = x
    method set x = contents <- x
  end;;
```

Exercice 16 (continued)

Parce que le paramètre `'a` apparaît à la fois en position positive dans la méthode `get` et négative dans la méthode `set`.

Exercice 16 (continued)

```
fun x -> (x : < a : int > cell :> < > cell);;
fun x -> (x : < > cell :> < a : int > cell);;
```

Exercice 16 (continued)

Si on cache la méthode set, le type `['a] cell` devient co-variant.

```
class ['a] cell_set x =
  object
    val mutable contents : 'a = x
    method get = x
    method private set x = contents <- x
  end;;
fun x -> (x : < a : int > cell_set :> < > cell_set);;
```

Inversement, si on cache la méthode `get`, le type `['a] cell` devient contra-variant.

```
class ['a] cell_get x =
  object
    val mutable contents : 'a = x
    method private get = x
    method set x = contents <- x
  end;;
fun x -> (x : < > cell_get :> < a : int > cell_get);;
```

Exercice 16 (continued)

Oui, car `'a` n'est pas le type de self, mais d'un paramètre. Il ne s'agit pas d'une méthode binaire. Un exemple sans intérêt, mais qui montre que tout se passe bien :

Exercice 17, page 16

En java, le type de `this` est celui d'un objet de la classe courante. La méthode `min` de la classe `Point` sera donc typée comme prenant un object de type `Point` et retournant un object de type `Point`. Dans la méthode `min` de la classe `Bipoint` l'expression `super.min(z)` aura le type `Point`. Il faut donc utiliser une conversion pour en faire un object de type `Bipoint`. Connaissant le code de la classe `Point` nous pouvons en déduire que cette coercion est sûre, mais cette déduction n'est pas possible à partir de la seule connaissance du type de la classe `Point`.

Bipoint.java

```
class Point {
    int x;
    Point (int x0) { x = x0; }
    int getx () { return x; }
    Point min (Point z) {
        if (this.getx() < z.getx()) return this; else return z;
    }
}

class Bipoint extends Point {
    int y;
    Bipoint (int x0, int y0) { super(x0); y = y0; }
```

```
    int gety () { return y; }
    Bipoint min (Bipoint z) {
        if (this.gety() < z.gety()) return this;
        else return (Bipoint)(super.min (z));
    }
}

class Main {
    public static void main (String arg[]) {
        Bipoint p = new Bipoint (1,2);
        Bipoint q = new Bipoint (2,1);
        Bipoint r = p.min(q);
        System.out.println(r.gety());
    }
}
```

Bipoint.java

Exercice 18, page 22

On ne peut pas chacher la méthode `code` dans la classe `codage` car elle doit être utilisée de l'extérieure sur l'argument de la méthode binaire `decode`. On la rend donc publique. Pour recouvrer une notion d'abstraction équivalente à celle du programme java, on utilise les modules ; plus précisément on rend le type retourné par la fonction `code` abtrait. La méthode `code` reste donc publique, mais la valeur qu'elle retourne ne peut pas être exploité puis que sont type est abstrait.

```
module M =
  (struct
    type t = int
    class codage n = object (_ : 'mytype)
      val code : int = n
      method code = code
      method decode (z : 'mytype) = (z#code = code)
    end
  end : sig
    type t
    class codage : int -> object ('mytype)
      method code : t
      method decode : 'mytype -> bool
    end
  end);;
class c = M.codage
let p = new c 1
and q = new c 2
let _ = p#decode q
    ;;
```

Exercice 21, page 14

```
class ['sujet] grande_fenêtre =
  object (self)
    inherit ['sujet] fenêtre
    val mutable taille = 1
    method retaille x =
      taille <- taille + x;
```

```
      self#envoie (fun obs -> obs#retaillé)
    val mutable devant = false
    method en_avant =
      devant <- true;
      self#envoie (fun x -> x#devant)
    method dessine =
      Printf.printf "{Position = %d; Taille = %d}" position taille;
  end;;
```

Exercice 21 (continued)

```
class ['sujet] grand_gestionnaire =
  object
    inherit ['sujet] gestionnaire
    method retaillé (s : 'sujet) : unit = s#en_avant
    method devant (s : 'sujet) : unit = s#dessine
  end;;

new grande_fenêtre#ajoute (new grand_gestionnaire);;
let w = new grande_fenêtre in
  w#ajoute (new grand_gestionnaire); w#déplace 1; w#retaille 2;;
```

{Position = 1; Taille = 1}{Position = 1; Taille = 3}– : unit = ()

Exercice 21 (continued)

```
class ['sujet] big_brother =
  object
    inherit ['sujet] inspecteur
    method retaillé (s:'sujet) = print_string "<R>"
    method devant (s:'sujet) = print_string "<D>"
  end;;
```

Et voilà le tout ensemble :

```
let w = new grande_fenêtre in
   w#ajoute (new grand_gestionnaire); w#ajoute (new big_brother);
   w#retaille 2; w#déplace 1;;
```

<R><D>{Position = 0; Taille = 3}<D>{Position = 1; Taille = 3}– : unit = ()

Exercice 25, page 8

$$
\begin{aligned}
&\boxed{(\lambda z.\lambda t.\lambda f.(z\ t)\ f)\ (\lambda x.\lambda y.x)}\ 1\ 2 \longrightarrow\\
&((\lambda t.\lambda f.(z\ t)\ f)[\lambda x.\lambda y.x/z])\ 1\ 2 \equiv\\
&\boxed{(\lambda t.\lambda f.(\lambda x.\lambda y.x)\ t\ f)\ 1}\ 2 \longrightarrow\\
&((\lambda f.((\lambda x.\lambda y.x)\ t)\ f)[1/t])\ 2 \equiv\\
&\boxed{(\lambda f.(\lambda x.\lambda y.x)\ 1\ f)\ 2} \longrightarrow\\
&(((\lambda x.\lambda y.x)\ 1)\ f)[2/f] \equiv\\
&(\boxed{(\lambda x.\lambda y.x)\ 1}\ 2) \longrightarrow\\
&((\lambda y.x)[1/x])\ 2) \equiv\\
&\boxed{(\lambda y.1)\ 2} \longrightarrow\\
&1[2/y] \equiv \boxed{1}
\end{aligned}
$$

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
```

Exercice 25 (continued)

Le terme $(\lambda x.\lambda y.y$ convient. En effet :

$$\begin{array}{l}\boxed{(\lambda z.\lambda t.\lambda f.(z\ t)\ f)\ (\lambda x.\lambda y.y)}\ 1\ 2 \longrightarrow \\ \dots \\ (\boxed{(\lambda x.\lambda y.y)\ 1}\ 2) \longrightarrow \\ ((\lambda y.y)[1/x])\ 2) \equiv \\ \boxed{(\lambda y.y)\ 2} \longrightarrow \\ y[2/y] \equiv 2\end{array}$$

On vérifie :

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

```
- : int = 2
```

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

Exercice 26, page 10

Le programme 1 2.

Exercice 26 (continued)

Le programme $(\lambda x.x\ x)\ (\lambda x.x\ x)$. En effet il se réduit en tête en une étape sur lui-même.

$$\begin{array}{l}\boxed{(\lambda x.x\ x)\ (\lambda x.x\ x)} \longrightarrow \\ ((x\ x)[\lambda x.x\ x/x]) \equiv \\ \boxed{(\lambda x.x\ x)\ (\lambda x.x\ x)} \longrightarrow \\ \dots\end{array}$$

Exercice 27, 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");;
```

Exercice 27 (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

Exercice 28, page 29

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

Exercice 28 (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.

Exercice 28 (continued)

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

Exercice 28 (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 };;
```

Exercice 28 (continued)

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

Exercice 29, 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});
  } ;;
```

Exercice 29 (continued)

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

– : int = 2

Exercice 29 (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`.

Exercice 30, page 31

En Java, il n'y a pas de solution bien typée : on est obligé de perdre de l'information et d'utiliser des coercions pour la récupérer. En Ocaml, la méthode `equal` est bien typée, mais asymmétrique. On doit rendre montrer une partie de la représentation (comme en Nice).