Représentation des objets (dans le modèle à enregistrement).

Didier Rémy
Octobre 2000

http://cristal.inria.fr/~remy/poly/mot/11/
http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/mot/11/

Cours	Exercises

Slide 1

Résumé

Slide 2

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 ?

Nous nous limitons au modèle à enregistrements. Nous considérons le cas général, et quelques cas particuliers intéressants.

Représentation des objets

Slide 3

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.

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). x)
let bump_point self = self .(0). x <- self.(0). getx self  + 1
let partagé =
  { abscisse  = 1; point = getx_point; bump = bump_point }
let new_point x : obj = [| partagé; x |]
```

Exemple

Envoi de message

```
let p = new point 17 in p#bump
```

Traduction

```
let p = new_point 17 in p.(0).bump p
```

Slide 6

Se réduit en

```
let p = [| meth; 17 |] in bump_point p
```

puis

```
[| meth; 18 |]
```

Enregistrements anonymes

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).

Slide 7

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 :

Slide 8

$$c \in [1, p] \times \ell \in [1, q] \rightharpoonup d \in I\!N$$

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

Slide 10

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.

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

Slide 11

On peut rendre la compaction 2 très efficace au prix d'une indirection supplémentaire en coupant $\ell \in [1, q] \mapsto d$ en

$$\ell_{fort} \in [1, q_1] \mapsto (\ell_{faible} \in [1, q_2] \mapsto d) \qquad \text{où } q_1, q_2 \approx \sqrt{q}$$

Beaucoup plus d'opportunités de compaction. Jamais de gros trous.

C'est la solution utilisée dans Ocaml.

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.

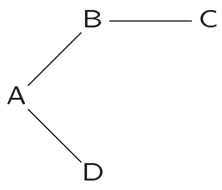
Ce cas est assez fréquent, surtout si on fait au préalable une analyse de flux.

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

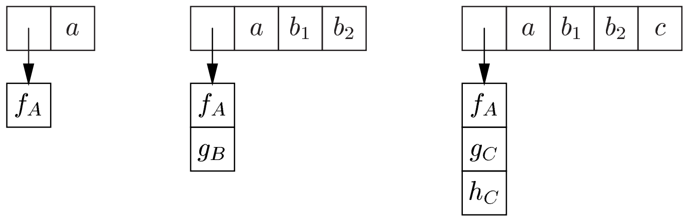
L'accès à une variable d'instance $p.x$ peut être optimisé lorsque la position de x est connue statiquement, c'est-à-dire dès que l'on connaît une classe parente de l'objet qui possède ce champ (1).

Dans l'exemple ci-dessus, un accès à la variable a dans un objet d'une sous-classe de B est uniformément $p.(1)$.

Slide 14

Accès aux méthodes

Cette optimisation s'étant aux méthodes (avec une indirection).



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 $p.(0).(1)\ p\ v_1\ v_2$.

Slide 15

Avantages et limitations

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

Slide 18

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.