DEA I_3
Module Fondements des Bases de données
Des Modèles à valeurs complexes aux Modéles semi-structurés en passant par les Modèles orientés-objet

Véronique Benzaken

October 22, 2001

Abstract

Présentation des modèles à valeur complexes et orientés-objet très largement inspirée de [AHV95]

Chapter 1

Valeurs complexes

1.1 Préliminaires : motivation et généralités

1.1.1 Motivation

L'avantage majeur du modéle relationnel réside en la très grande simplicité des concepts sous-jacents : notion de relation (table). Dans ce modèle, le schéma d'une relation est composé d'un ensemble d'attributs prenant leurs valeurs dans des domaines élémentaires (ou atomiques). C'est la contrainte de première forme normale (1NF). Or, une telle contrainte inhérente au modèle impose de sévères limitations en terme de modélisation. Afin de dépasser une telle limite, le modèle à valeurs complexes (ou valeurs structurées, ou objets complexes) a été introduit.

1.1.2 Généralités

Intuitivement, une valeur complexe est une (instance de) relation dont les valeurs de certains (voire tous) de leurs attributs peuvent également être des (instances de) relations.

Dans le modèle relationnel, la structure des des données peut être vue comme le résultat de l'application, à des valeurs atomiques, de deux constructeurs. Le constructeur de *n-uplet* pour construire des n-uplets. Le constructeur d' *ensembles* pour construire des ensembles de n-uplets (les relations).

Dans le modèle à valeurs complexes, ces constructeurs peuvent être appliqués récursivement et de façon arbitraire. Par conséquent, une valeur complexe peut être vue comme un arbre infini dont les noeuds internes sont étiquetés par un constructeur et les feuilles par une valeur atomique.

Au niveau du schéma, nous spécifierons un ensemble de *sortes* ou *types* complexes pour indiquer la structure des données et au niveau instance nous peuplerons ces types.

Exemples

type	*valeur*
dom	a
{**dom** }	$\{a, b, c\}$
⟨ A : **dom**, B : **dom** ⟩	⟨ A : a, B : b ⟩
{⟨ A : **dom**, B : **dom** ⟩}	⟨ A : a, B : b ⟩}
{ { **dom** } }	{ { a, b }, { a }, {} }
{ ⟨ A: **dom**, B : **dom**, C : { ⟨ A : **dom**, E : { **dom** } ⟩ ⟩ }	{ ⟨ A: a, B : b, C : { ⟨ A : c, E : { } ⟩ , ⟨ A : d, E : { } ⟩ },
	⟨ A : e, B: f, C : {} ⟩ }

Nous présentons dans le reste de ce chapitre une algèbre et un calcul équivalents pour les modèles à valeurs complexes. L'accent sera porté sur les constructeurs de n-uplets et d'ensemble même si d'autres constructeurs ont été ultérieurement introduits (listes, bags et union).

1.2 Bases de données à valeurs complexes

Nous nous donnons les ensembles suivants : **relname**, **att**, **dom** qui dénotent respectivement les noms de relations, les noms d'attributs et les constantes.

Definition 1 L'ensemble $\mathcal{T}$ des types τ est défini par :

$$\tau ::= \textbf{dom} \mid \langle\ \mathrm{B}_1 : \tau, \ldots, \mathrm{B}_k : \tau\ \rangle \mid \{\ \tau\ \}$$

où $k \geq 0$, et les B_i sont deux à deux distincts. Intuitivement, un élément de **dom** est une constante (**dom** désigne des types élémentaires (entiers, booléens, etc...) ; un élément de $\langle\ \mathrm{B}_1 : \tau, \ldots, \mathrm{B}_k : \tau\ \rangle$ est un k-uplet dont chaque composant B_i est de type τ_i enfin un élément de $\{\ \tau\ \}$ est un ensemble *fini* d'éléments de type τ.

Definition 2 Plus formellement, l'interprétation, notée $[\![]\!]$ d'un type τ est définie par :

1. $[\![\ \textbf{dom}\ ]\!] = \textbf{dom}$
2. $[\![\ \{\ \tau\ \}\ ]\!] = \{\ \{\ v_1, \ldots, v_j\ \}\ ; j \geq 0, v_i \in [\![\tau]\!], i \in [1, j]\ \}$
3. $[\![\langle\ \mathrm{B}_1 : \tau_1, \ldots, \mathrm{B}_k : \tau_k\ \rangle]\!] = \{\ \langle B_1 : v_1, \ldots, B_k : v_k \rangle\ ; v_j \in [\![\tau_j]\!], j \in [1, k]\ \}$

Etant donné un type τ, une instance du type τ, c'est à dire un élément de $[\![\tau]\!]$ est appelé *une valeur complexe*.

Remarque 1 Une valeur complexe donnée peut avoir plusieurs types distincts (ensembles vides ;-)

L'algèbre relationnelle opére avec des ensembles de n-uplets, similairement l'algèbre à valeur complexes opérera avec des *ensembles de valeurs complexes*.

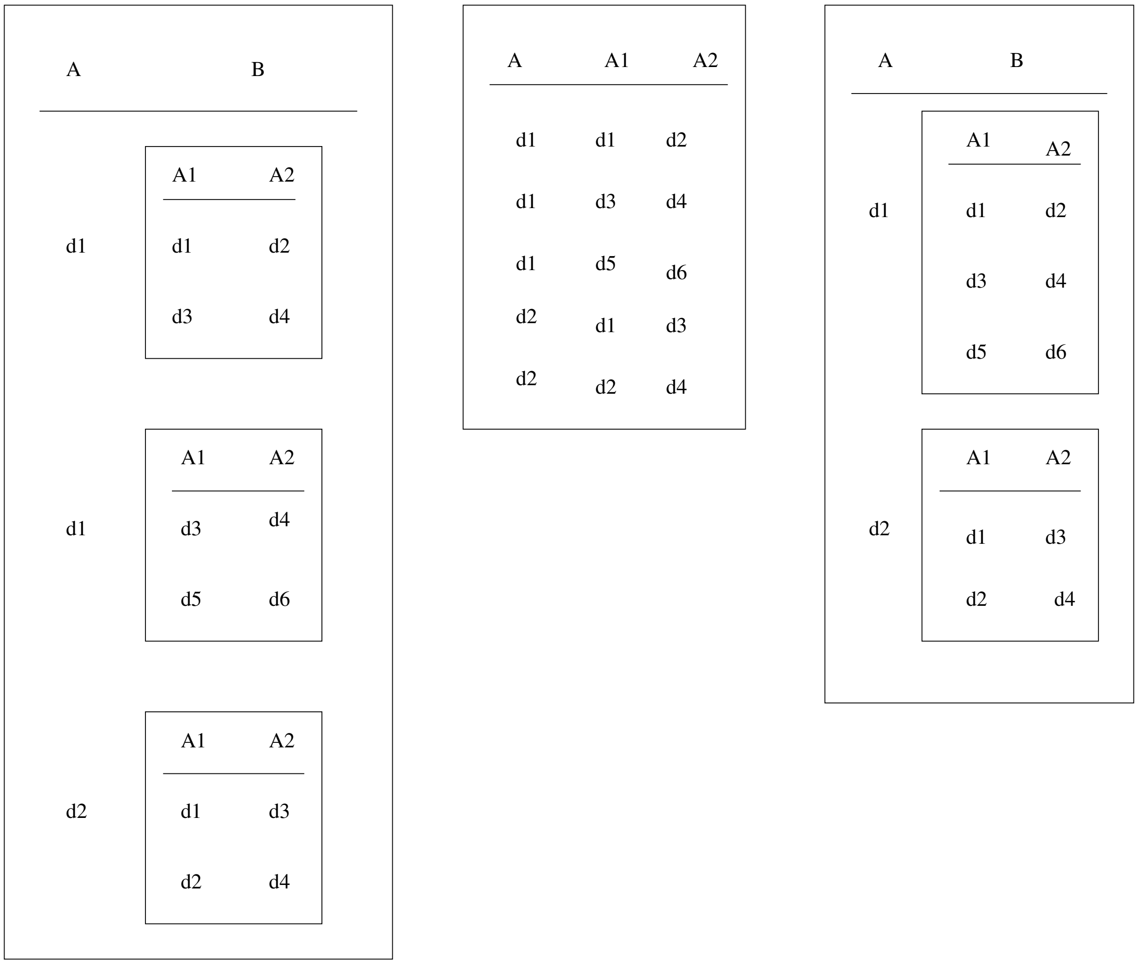
Definition 3 Un relation complexe de type τ est un ensemble fini de valeurs complexes instances du type τ.

Enfin, nous supposons l'existence d'une fonction (notée **type**) de l'ensemble **relname** vers les types $\mathcal{T}$. Nous supposons en outre que pour chaque type τ, il existe une infinité de relations R possédant ce type.

Definition 4 [Schémas et Instances]
Un schéma est un élément de **relname**, et un schéma de base de données est un ensemble fini d'éléments de **relname**. Une *relation* sur R est un ensemble fini d'éléments de type **type**(R), soit donc un ensemble fini de $[\![\ \textbf{type}(R)\ ]\!]$. Enfin, une instance **I** d'un schéma **R** est une fonction qui est telle que pour chaque R de **R**, **I**(R) soit une relation sur R.

Exemple 1
type(R_1) = **type**(R_3) = $\langle A : \textbf{dom}, B : \{\ \langle A_1 : \textbf{dom}, A_2 : \textbf{dom}\ \rangle\ \}\ \rangle$

type(R_2) = $\langle A : \textbf{dom}, A_1 : \textbf{dom}, A_2 : \textbf{dom} \rangle$



I(R1) I(R2) I(R3)

Figure 1.1: Exemple : valeurs complexes

1.3 Algèbre à valeurs complexes

Comme dans le cas de l'algèbre relationnelle, nous pouvons considérer que l'algèbre à valeurs complexes est un langage fonctionnel basé sur un certain nombre d'opérateurs. Par la suite, nous noterons ALG^{cv} ce langage.

1.3.1 Le coeur de ALG^{cv}

Soient $I, I_1, I_2, \ldots$ des relations de types τ, $\tau_1, \tau_2 \ldots$ respectivement. Notons qu'une relation de type τ est un *ensemble* de valeurs de type τ.

- *opérations ensemblistes*
  Si $\tau_1 = \tau_2$ alors $I_1 \bigcap I_2, I_1 \bigcup I_2, I_1 \setminus I_2$, sont des relations de type $\tau_1$et leurs valeurs sont définies de manière standard.

- *opérations n-uplet*
  Si I une relation de type $\tau = \langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle$, alors

  – $\sigma_\gamma(I)$ est une relation de type τ.
  La condition de sélection γ est de la forme (modulo restriction de typage) $B_i = d$, $B_i = B_j$, $B_i \in B_j$ ou $B_i = B_j.C$, où d est une constante, et dans le dernier cas τ_j est un type n-uplet ayant un attribut C. Nous avons :

  $$\sigma_\gamma(I) = \{v; v \in I, v \models \gamma\}$$

  où $\models$ est définie par :

  * $\langle \ldots, B_i : v_i, \ldots \rangle \models B_i = d$ si $v_i = d$
  * $\langle \ldots, B_i : v_i, \ldots, B_j : v_j, \ldots \rangle \models B_i = B_j$ si $v_i = v_j$
  * $\langle \ldots, B_i : v_i, \ldots, B_j : v_j \rangle \models B_i \in B_j$ si $v_i \in v_j$
  * $\langle \ldots, B_i : v_i, \ldots, B_j : \langle \ldots, C : v_j, \ldots \rangle, \ldots \rangle \models B_i = B_j.C$ si $v_i = v_j$

  – $\pi_{B_1,\ldots,B_l}(I), l \leq k$ est une relation de type $\langle B_1 : \tau_1, \ldots, B_l : \tau_l \rangle$ avec
  $\pi_{B_1,\ldots,B_l}(I) = \{\langle B_1 : v_1, \ldots, B_l : v_l \rangle; \exists v_{l+1}, \ldots, v_k, \langle B_1 : v_1, \ldots, B_l : v_l, B_{l+1} : v_{l+1}, \ldots, B_k : v_k \rangle \in I\}$

- *opérations constructives*

  – *powerset(I)* est une relation de type $\{\tau\}$ et

  $$\textit{powerset(I)} = \{v; v \subseteq I\}.$$

  – Si $A_1, \ldots, A_n$ sont des attributs distincts, *tup-create*$_{A_1,\ldots,A_n}(I_1, \ldots, I_n)$ est de type $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$ et

  $$\textit{tup-create}_{A_1,\ldots,A_n}(I_1, \ldots, I_n) = \{\langle A_1 : v_1, \ldots, A_n : v_n \rangle; \forall i, (v_i \in I_i)\}$$

  – *set-create(I)* est de type $\{\tau\}$ et *set-create(I)* = $\{I\}$

- *opérations destructives*

  – Si $\tau = \{\tau'\}$, alors *set-destroy(I)* est de type τ' et *set-destroy(I)* = $\bigcup I = \{w; \exists v \in I, w \in v\}$
  – Si I est de type $\langle A : \tau' \rangle$, *tup-destroy(I)* est de type τ' et *tup-destroy(I)* = $\{v; \langle A : v \rangle \in I\}$

Nous sommes désormais en mesure de définir le langage ALG^{cv}. Soit **R** un schéma de base de données. Une requête renvoie un ensemble de valeurs de même type. Nous appelons ce type le type de la requête. Par analogie avec le modèle relationnel, une requête de type τ renvoie un ensemble de valeurs de type τ. Les requêtes sont définies comme suit :

Definition 5 [Requêtes]

- *base* Chaque nom de relation R de **R** est une requête algèbrique de type **type**(R). La réponse à la requête esat **I**(R).
- *constantes* Pour tout élément a, $\{a\}$ est une requête algébrique de type **dom**. La réponse à la requête est simplement $\{a\}$.
- Soient $q_1, q_2, \ldots$, des requêtes, soit γ une condition de sélection et $A_1, \ldots$ des attributs.
$q_1 \bigcap q_2, q_1 \bigcup q_2, q_1 \backslash q_2, \sigma_\gamma(q_1), \pi_{A_1,\ldots,A_k}(q_1), tup{-}create_{A_1,\ldots,A_k}(q_1, \ldots, q_k), powerset(q_1), tup{-}destroy(q_1), set{-}destroy(q_1), set-create(q_1)$ sont des requêtes modulo règles de typage.

Le type d'une requête et de sa réponse sont définis de manière évidente (laissé en exercice).

Exemple 2 Reprenons l'instance **I** de l'exemple précédent. Soient les requêtes :

$$\begin{array}{l}
J_1 = \sigma_{A=d_2}(R_1), \\
J_2 = \pi_B(J_1), \\
J_3 = tup - destry(J_2), \\
J_4 = set - destroy(J_3), \\
J_5 = powerset(J_4), \\
J_6 = tup - create_C(J_4)
\end{array}$$

1.4 Calcul à valeurs complexes

Nous présentons l'équivalent dans le domaine des modèles à valeurs complexes des calculs à variable n-uplet ou domaine du modèle relationnel. Nos nommons ce calcul CALC^{cv}. Pour chaque type, nous supposons l'existance d'un ensemble infini dénombrable de *variables* de ce type. Une variable est *atomique* si son type est **dom**. Soit **R** un schéma. Les *termes* sont définis par :

Definition 6 [termes]
$t ::= c \mid x \mid x.A$ où c dénote une constant, x une variable et A un nom d'attribut.

Les formules atomiques sont définies par :

Definition 7 $fa ::= R(t) \mid t = t' \mid t \in t' \mid t \subseteq t'$ où $R \in$ **R**, t, t' sont des termes.

Les formules sont construites usuellement à partir des formules atomiques au moyen des connecteurs logiques $\wedge, \vee, \neg, \forall, \exists$.

Definition 8 [requête]
Une requête est une expression de la forme : $\{x; \phi\}$ où x est l'unique variable libre dans la formule ϕ.

1.5 Théorèmes d'équivalence

Comme en relationnel nous trouvons la notion de calcul indépendant du domaine ainsi que de requêtes saines. Pour plus de détail consulter [AHV95].

Theorem 1 ALG^{cv} et le calcul à valeurs complexes indépendant du domaine sont équivalents.

Chapter 2

Modèles Orientés-objet

2.1 Présentation informelle

Les bases de données orientées-objet sont au carrefour de trois mondes :

- Le domaine des modèles à objet structurés,
- Le domaine des modèles sémantiques et
- Le domaine de la programmation orientée-objet

À ce jour, il n'existe pas un unique modèle orienté objet comme il existe un seul modèle relationnel. Nous concentrerons notre étude sur les sept concepts les plus importants des modèles objet :

1. objets et identificateurs d'objets,
2. valeurs structurées et types,
3. classes,
4. méthodes,
5. hiérarchies de spécialisation,
6. héritage et liaison tardive,
7. encapsulation

Un des points faibles du modèle relationnel réside dans le fait que l'on soit obligé d'utiliser des valeurs atomiques (entiers, chaînes, etc) souvent appelées clefs, pour référencer (ou distinguer) des entités ou objets-du-monde-réel. À titre d'exemple, supposons que le nom et le prénom d'une personne soient utilisés comme clef. Du point de vue de la gestion physique (système) utiliser ceci pour identifier la personne est lourd (beaucoup d'octets sont nécessaires pour représenter le nom et le prénom).

Un inconvénient plus sensible apparaît si le nom de la personne change (en cas de marriage). Dans une telle circonstance, il y a une rupture dans la représentation de l'entité. En outre, il faudra propager la mise à jour à tous les n-uplets qui référencent cette personne.

Les modèles orientés-objet permettent la représentation explicite des entités tant sur le plan physique que conceptuel au travers de la notion *d'identité d'objet*. Conceptuelle-ment, un unique identificateur (oid) est assigné à chaque objet présent dans la base de données. Cette association est fixe au cours du temps, même si des attributs de l'objet se voient modifiés. Cette notion offre la possibilité de partager de l'information de manière harmonieuse. Un objet o peut être partagé par plusieurs autres objets simplement en référençant l'oid de o. Ceci est particulièrement important lors de mises à jour, ces dernières n'ont plus à être propagées.

Dans les modèles objet, une valeur structurée est généralement associée à chaque objet. Cette valeur peut être composée de valeurs atomiques et ou d'oid (i.e., de références au même ou à d'autres objets). Pour l'exemple 3, tout élément de la classe Film est associé à une valeur structurée dont le deuxième composant (acteur) est lui même un ensemble d'oid correspondants aux acteurs du film.

Exemple 3

```
(* Definition du schema et de la base *)

       create schema SchemaPariscope;
       create base   BasePariscope;

(* Definition des classes *)

       class Personne
             type tuple(nom: string, nationalite: string, sexe: string);
       class Realisateur inherit Personne
             type tuple(realisations: {Film});
       class Acteur inherit Personne
             type tuple(joue-dans: {Film},
                    recompenses: set(tuple(prix: string, annee: integer)));
       class Acteur-Realisateur inherit Realisateur, Acteur
       class Film
             type tuple(titre: string,
                        acteurs: set(cteur),
                        realisateur: Realisateur);
       class Cinema
             type tuple(nom: string, adresse: string, tel: string);

(* Definition des racines de persistance *)

       name Pariscope:            set(tuple(cinema: Cinema,
                                         heure: string,
                                         prix: integer,
                                         film: Film));
       name PersonnesAimees:      set(Personne);
       name ActeursPreferes:      set(Acteur);
       name RealisateurPrefere:   Realisateur;

(* Definition des methodes *)

       method lire-nom in class Person : string
              { if (sexe = ''masculin'')
                   return ''Mr.'' + self.name;
                else
                   return ''Mme'' + self.name}

       method lire-nom in class Realisateur : string
              { return ''Realisateur'' + self.name };

       method lire-nom in class Acteur-Realisateur : string
              { return ''Realisateur'' + self.name };
```

Comme dans le cas des modèles sémantiques, des objets de même type sont regroupés dans des *classes*. Dans les modèles objets, les classes sont également utilisées comme unités de définition des traitements. En pratique, ceci est réalisé en associant à chaque classe un ensemble de *méthodes*. Une méthode peut être simple (par exemple retourner la valeur du nom) ou arbitrairement complexe. Une méthode possède un nom, une signature ainsi qu'une implémentation. Le nom et la signature servent d'interface externe à la méthode. L'implémentation de la méthode est de manière générale écrite dans un langage de programmation. Le choix du langage ne fait pas partie du modèle de données.

Les classes, comme dans les modèles sémantiques, peuvent être organisées sous forme d'une hiérarchie d'héritage. Dire qu'une classe c est sous classe de c' (noté $c \leq c'$) induit les conséquences suivantes :

1. Le type des valeurs associées à c doit être un sous-type de celui des valeurs associées à c',
2. S'il existe une méthode de nom m associée à c' alors une méthode de même nom doit exister pour c. Dans certains cas l'implémentation de m pour c peut être la même que pour c', la méthode est alors héritée et il n'est pas nécessaire de redéfinir son implémentation. Lorsque l'implémentation de m pour c est différente de celle de c', on dit que l'implémetation de m pour c masque l'implémentation de m pour c'. Le choix de l'implémetation à utiliser s'appelle la résolution, cette résolution si elle est opérée à l'exécution s'appelle résolution tardive.

Un dernier concept important, relatif aux modèles orientés-objet est celui *d'encapsulation*. Cette notion est issue du domaine des types abstrait de données. Cette notion permet d'établir une claire séparation entre la manière dont les objets sont manipulés et la manière dont ils sont physiquement stockés. Cette notion est plus claire si l'on songe au fonctionnement traditionnel d'une base de données dans lequel on distingue deux modes d'utilisation : le mode administrateur du système (qui crée et met à jour le schéma) et le mode utilisateur (qui opère des requêtes et des mises à jour sur les données).

En mode utilisateur, le principe d'encapsulation impose que les seuls accès possibles aux objets se fasse au travers de l'utilisation des méthodes. Dans certains modèles orientés-objet, ce dernier principe est levé dans certains cas très précis tels que l'utilisation du langage de requêtes.

2.2 Définition formelle d'un modèle orienté-objet

Cette section présente la définition formelle d'un modèle particulier. Cette présentation emprunte beaucoup aux modèles IQL [AK89] et O_2 [LRV88, LR89a, LR89b]. La plupart des modèles orientés-objet possèdent les caractéristiques décrites dans cette section.

2.2.1 Types et hiérarchies de classes

Les définitions formelles d'objet, de type et de hiérarchies de classes sont étroitement liées. Un objet consiste en la donnée d'une paire (*identificateur*, *valeur*). Les identificateurs sont éléments d'un domaine particulier appelé OID. Les valeurs sont ce que l'on appelle des valeurs structurées.

Nous supposons l'existence de types atomiques, prenant leur valeurs dans les domaines suivants deux à deux disjoints: **integer**, **string**, **bool**, **float**. L'union de ces domaines est notée **dom** et représente l'ensemble des valeurs atomiques. Les éléments de **dom** sont appelés des *constantes*. Nous supposons également l'existence d'un ensemble infini **obj** = $\{o_1, o_2, \ldots\}$ *d'identificateurs d'objets* d'un ensemble **class** de noms de classes et d'un ensemble **att** de noms d'attributs. Un identificateur spécial, noté *nil* représentera la valeur indéterminée (null). Étant donné un ensemble O d'OIDs, la famille des valeurs sur O est définie par :

Definition 9

1. *nil*, tout élément de **dom** et tout élément de O sont des valeurs sur O et
2. si $v_1, \ldots, v_n$ sont des valeurs sur O et si $A_1, \ldots, A_n$ sont des noms d'attributs distincts alors le n-uplet $[A_1 : v_1, \ldots, A_n : v_n]$ et l'ensemble$\{v_1, \ldots, v_n\}$ sont des valeurs sur O.

L'ensemble de toutes les valeurs sur O est noté **val**(O). Un objet est donc une paire (o, v) où o est un élément de OID et v une valeur (comme définie ci dessus).

En général, les modèles orientés-objet incluent d'autre constructeurs tels que les listes et les multi-ensembles, nous ne les considérerons pas ici. L'exemple 4 illustre la notion de valeur et d'objet.

Exemple 4

```
Quelques valeurs
       [cinema : oid7, heure : ‘‘16h45’’, prix : 45, film : oid22]

       {‘‘G. Massina’’, ‘‘S.Loren’’, ‘‘M.Mastroianni’’}

       [titre       : ‘‘La Strada’’,
        realisateur : ‘‘F. Fellini’’,
        acteurs     : {oid81, oid198, oid243, oid34}]
Un objet
       (oid22, [titre       : ‘‘La Strada’’,
                realisateur : ‘‘F. Fellini’’,
                acteurs     : {oid81, oid198, oid243, oid34}])
```

Les objets sont regroupés dans des classes. Tous les objets regroupés dans une même classe possèdent des valeurs de même type. Le type correspondant à chaque classe est spécifié dans le schéma. Les types sont définis par rapport à un ensemble C donné de noms de classes, la famille des types sur C est définie de la manière suivantes :

Definition 10

1. **integer**, **float**, **bool**, **string** sont des types,
2. les noms de classe de C sont des types,
3. si τ est un type alors$\{\tau\}$ est un type (ensemble),
4. si $\tau_1, \ldots, \tau_n$ sont des types, si $A_1, \ldots, A_n$ sont des noms d'attributs distincts, alors $[A_1 : \tau_1, \ldots, A_n : \tau_n]$ est un type (n-uplet)

L'ensemble des types sur C auquel on ajoute le nom de classe **any** (la racine de toutes les classes) est noté **type**(C). La classe **any** *ne peut pas apparaître* dans la construction d'autres types. L'exemple 5 illustre la notion de type pour le schéma de l'exemple 3.

Exemple 5

```
Un exemple de type

          [nom: string, nationalite: string, sexe: string]

Il est possible de donner des noms aux types :

Quelques types nommes

          TypePersonne    = [nom        : string,
                             nationalite: string,
                             sexe       : string]

          TypeRealisateur = [nom        : string,
                             nationalite: string,
                             sexe       : string,
                             realise    : {Film}]

          TypeCinema      = [nom        : string,
                             adresse    : string,
                             tel        : string]

          TypePariscope   = [cinema     : Cinema,
                             heure      : string,
                             tarif      : integer,
                             film       : Film]

          TypeFilm        = [titre      : string,
                             acteurs    : {Acteur},
                             realisateur: Realisateur]

          TypePrix        = [prix       : string,
                             annee      : integer]
```

Dans un schéma orienté-objet, à chaque nom de classe c est associé un type (noté $\sigma(c)$) qui indique quel sera le type des objets de cette classe. Pour chaque objet (o, v) de la classe c, v doit avoir la structure décrite par $\sigma(c)$.

Les classes sont organisées selon une hiérarchie de spécialisation. Une telle hiérarchie de classes possède trois composants :

1. un ensemble de classes,
2. les types associés à ces classes et
3. une spécification de l'ordre de spécialisation.

Formellement, une hiérarchie de classe est un triplet $(C, \sigma, \prec)$ où C est un ensemble fini de noms de classes, σ une fonction de C vers **types**(C), et $\prec$ une relation d'ordre partiel sur C. Informellement, le type associé à une sous-classe doit être un "raffinement" du type associé à sa (ses) super-classe(s). Par exemple, on s'attend à ce que la classe Étudiant précise les informations relatives à la classe Personne en ajoutant des attributs. Afin de prendre en compte cette notion, nous utilisons une relation de sous-typage ($\leq$) qui permet de spécifier qu'un type en spécialise un autre.

Definition 11 Soit $(C, \sigma, \prec)$ une hiérarchie de classes. La relation de sous-typage sur **types**(C) est le plus petit ordre partiel sur **types**(C) qui satisfait les conditions suivantes :

1. si $c \prec c'$ alors $c \leq c'$,

2. si $\tau_i \leq \tau'_i$ pour $i \in [1, n]$ et $n \leq m$ alors
$[A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_m : \tau_m] \leq [A_1 : \tau'_1, \ldots, A_n : \tau'_n]$

3. si $\tau \leq \tau'$ alors $\{\tau\} \leq \{\tau'\}$ et

4. pour tout τ, $\tau \leq$ **any**

Une hiérarchie est dite *bien formée* si pour toute paire de classes (c, c'), $c \prec c'$ implique que $\sigma(c) \leq \sigma(c')$.
Ainsi, pour l'exemple 5, nous avons $\texttt{TypeRealisateur} \leq \texttt{TypePersonne}$ et $\texttt{TypeRealisateur} \not\leq \texttt{TypeFilm}$. La hiérarchie qui serait obtenue en imposant la contrainte : $\texttt{Realisateur} \prec \texttt{Film}$ ne serait pas bien formée.
Nous ne considérerons dans la suite que des hiérarchies bien formées. Reprenons la hiérarchie de classes $(C, \sigma, \prec)$ de l'exemple 3. l'ensemble des classes est :

$$C = \{\texttt{Personne}, \texttt{Realisateur}, \texttt{Acteur}, \texttt{ActeurRealisateur}, \texttt{Cinema}, \texttt{Film}\}.$$

l'ordre $\prec$ (les contraintes définies pour ces classes) est défini par :

$$\begin{array}{lcl}
\texttt{Acteur} & \prec & \texttt{Personne} \\
\texttt{Realisateur} & \prec & \texttt{Personne} \\
\texttt{ActeurRealisateur} & \prec & \texttt{Acteur} \\
\texttt{ActeurRealisateur} & \prec & \texttt{Realisateur}
\end{array}$$

Enfin, reprenant les définitions des types `TypePersonne, TypeActeur` ... etc., de l'exemple 5, nous avons :

$$\begin{array}{lcl}
\sigma(\texttt{Personne}) & = & \texttt{TypePersonne} \\
\sigma(\texttt{Cinema}) & = & \texttt{TypeCinema} \\
\sigma(\texttt{Film}) & = & \texttt{TypeFilm} \\
\sigma(\texttt{Realisateur}) & = & \texttt{TypeRealisateur} \\
\sigma(\texttt{Acteur}) & = & [\texttt{nom} : \mathbf{string}, \ \texttt{nationalite} : \mathbf{string} \\
 & & \texttt{sexe} : \ \mathbf{string}, \ \texttt{joue}-\texttt{dans} : \{\texttt{Film}\}, \\
 & & \texttt{recompenses} : \ \texttt{TypePrix}] \\
\sigma(\texttt{ActeurRealisateur}) & = & [\texttt{nom} : \mathbf{string}, \ \texttt{nationalite} : \mathbf{string} \\
 & & \texttt{sexe} : \ \mathbf{string}, \ \texttt{joue}-\texttt{dans} : \{\texttt{Film}\}, \\
 & & \texttt{recompenses} : \ \texttt{TypePrix}, \ \texttt{realise} : \{\texttt{Film}\}]
\end{array}$$

L'utilisation de nom de types est purement syntaxique, nous aurions pu, aussi bien, utiliser leur valeur correspondante. Nous aurions obtenu le même schéma. Notons enfin que $\sigma(\texttt{Realisateur}) \leq \sigma(\texttt{Personne})$ et que
$\sigma(\texttt{Acteur}) \leq \sigma(\texttt{Personne})$ etc.

2.2.2 Sémantique structurelle d'une hiérarchie de classes

Nous décrivons comment des valeurs peuvent être associées aux classes et types d'une hiérarchie. Comme les valeurs du modèle orienté-objet peuvent inclure des oid, la sémantique des types et des classes doit être définies simultanément. Afin de pouvoir donner ces définitions, nous nous baserons sur la notion centrale *d'assignation d'oid* qui permet d'associer à chaque classe un ensemble d'oids.

Definition 12 Soit $(C, \sigma, \prec)$ une hiérarchie (bien formée) de classes. Une *assignation d'oid* est une fonction π qui à chaque nom dans C fait correspondre un ensemble disjoint d'oid. Étant donnée une assignation d'oid π, *l'extension disjointe* de c est $\pi(c)$ et *l'extension* de c, notée $\pi^*(c)$ est $\cup\{\pi(c'); c' \in C, c' \prec c\}$

Étant donné une assignation d'oid π, si $c \prec c'$ alors $\pi(c) \subseteq \pi(c')$. Ceci traduit le fait qu'un objet de la classe c est aussi un objet de la classe c'. Du point de vue du typage, ceci exprime le fait que le opérations qui sont applicables aux objets de c' peuvent également être appliquées aux objets de c. En d'autres termes, un objet de c pourra être utilisé dans tout contexte où un objet de c' est attendu.

La sémantique des types peut être à présent définie, cette définition se fait relativement à une hiérarchie de classe donnée ainsi quà une assignation d'oids donnée.

Definition 13 Soient $(C, \sigma, \prec)$ une hiérarchie de classes et π une assignation d'oids. Soit $O = \cup\{\pi(c); c \in C\}$. Posons $\pi(\mathbf{any}) = O$. L'*interprétation disjointe* d'un type τ, notée *dom*(τ), est donnée par :

1. pour tout type atomique τ, *dom*(τ) est l'interprétation habituelle pour ce type.
2. *dom*(**any**) = **val**(O).
3. pour tout $c \in C$, *dom*(*c*) = $\pi^*(c) \cup \{nil\}$.
4. *dom*$(\{\tau\})$ = $\{\{v_1, \ldots, v_n\}; n \geq 0, v_i \in dom(\tau), i \in [1, n]\}$ et
5. *dom*$([A_1 : \tau_1, \ldots, A_n : \tau_n])$ = $\{[A_1 : v_1, \ldots, A_n : v_n]; v_i \in dom(\tau_i), i \in [1, n]\}$.

2.2.3 Comportement : les méthodes

Le dernier composant, et non le moindre, des modèles orienté-objet est celui de méthode. Une méthode est constituée de :

1. un *nom*,
2. une *signature* et
3. une *implémentation*.

Il n'est pas difficile de définir le nom et la signature d'une méthode. En revanche, pour en spécifier l'implémentation, un langage de programmation est nécessaire. Dans le modèle orienté-objet, générique, présenté ici, le choix de tel ou tel autre langage de programmation n'est pas abordé. Un tel choix ne faisant pas véritablement partie du modèle. Nous reviendrons ultérieurement sur cet aspect.
Nous ne considérerons à présent que les noms et signatures des méthodes, seuls ceux-ci sont spécifiés au niveau du schéma. Nous supposons l'existence d'un ensemble **meth** de nom de méthodes. Soit $(C, \sigma, \prec)$ une hiérarchie de classes. Pour le nom de méthode m, une *signature* de m est une expression de la forme :

$$m : c \times \tau_1 \times \ldots \times \tau_{n-1} \rightarrow \tau_n$$

où c est un nom de classe de C, et chaque τ_i un type sur C. Cette signature est associée à la classe c, on dit alors que m *s'applique* aux objets de la classe c ainsi qu'aux objets de classes héritant m de c. Il est fréquent d'avoir, pour un même nom de méthode m différentes signatures relatives à différentes classes. Évidemment, des restrictions doivent être imposées (ces restrictions seront définies plus tard) sur la définition de telles signatures.

Nous présentons, au moyen d'exemple les notions de surcharge, d'héritage et de résolution tardive. Considérons les méthodes définies dans l'exemple 3. Toutes trois ont le même nom : `lire-nom`. Les signatures correspondantes sont :

$$\begin{array}{lcll} \mathtt{lire-nom} & : & \mathtt{Personne} & \rightarrow \mathbf{string} \\ \mathtt{lire-nom} & : & \mathtt{Realisateur} & \rightarrow \mathbf{string} \\ \mathtt{lire-nom} & : & \mathtt{ActeurRealisateur} & \rightarrow \mathbf{string} \end{array}$$

Les implémentations sont :

```
method lire-nom in class Personne : string
       { if (sexe = ``masculin'')
            return ``Mr.'' + self.name;
         else
            return ``Mme'' + self.name}

method lire-nom in class Realisateur : string
       { return ``Realisateur'' + self.name };

method lire-nom in class Acteur-Realisateur : string
       { return ``Realisateur'' + self.name };
```

La méthode `lire-nom` ayant trois implémentations différentes, on parle alors de *surcharge* du nom de la méthode. Précisément, l'implémentation de `lire-nom` dans la classe `Realisateur` *masque* celle de la classe `Personne`.
La classe `Acteur` est une sous-classe de la classe `Personne`. La méthode `lire-nom` devrait être également applicable aux objets de la classe `Acteur`. C'est effectivement le cas, on dit alors, que la classe `Acteur` hérite la méthode `lire-nom` de la classe `Personne`. Plus généralement, étant données deux classes c et c', telles que la méthode m soit définie dans c, $c' \prec c$ et qu'il n'existe pas de classe c'', $c' \prec c''$, $c'' \prec c$ et telle que m soit définie dans c'' on dit alors que c' hérite de la méthode m de c.
En particulier, la signature de la méthode m dans c' est identique à celle de m dans c sauf sur le premier argument où c est remplacé par c'. Pour notre exemple, la signature de la méthode `lire-nom` de la classe `Acteur` est : .

$$\mathtt{lire-nom} \quad : \quad \mathtt{Acteur} \quad \rightarrow \quad \mathbf{string}$$

Son implémentation est la même que celle de `Personne`. Déterminer quelle implémentation doit être utilisée pour une méthode m et une classe donnée c s'appelle la *résolution de méthode*, l'implémentation sélectionnée s'appelle la résolution de m pour c.
Supposons que nous ayons l'assignation d'oid π et que oid25 soit dans l'extension $\pi^*(\mathtt{Personne})$ du nom de classe `Personne`, supposons que la méthode `lire-nom` soit appelée avec l'oid25. Quelle implémentation choisir pour `lire-nom`? Deux politiques de sélection existent : la liaison statique ou la liaison dynamique.
Avec la *liaison statique*, l'implémentation choisie sera celle qui correspond au type (à la classe) défini(e) pour la variable contenant oid25. Ce type est obtenu lors de la compilation. Si oid25 est effectivement un élément d'une sous-classe, lors de l'appel, `Realisateur` par exemple, et bien c'est l'implémentation de `Personne` qui sera malgré tout utilisée avec la liaison statique.
Avec la *liaison dynamique* , le choix de l'implémentation à utiliser se fait à l'exécution. L'implémentation choisie correspond à celle qui est définie pour la classe de l'objet dénoté par la variable lors de l'appel effectif. Ainsi, contrairement à la liaison tardive, l'implémentation de `lire-nom` qui dans ce cas sera choisie est celle de la classe `Realisateur`.
En terminologie des langages orientés-objet, le nom de la méthode est souvent appelé *message*, l' appel de la méthode, $m(o, v_1, \ldots, v_{n-1})$, est appelé *envoi de message*, o, le *receveur* du message.
Certains langages n'utilisent, pour sélectionner l'implémentation, que le type du receveur, alors d'autres langages (ex., CommonLoops) utilisent également le type de tous les paramètres. Cette dernière manière d'opérer la sélection permet d'obtenir la propriété de *type safety* (c.f., cours du module M_4).
L'ensemble des méthodes applicables à un objet est appelée l'interface de l'objet. Le principe *d'encapsulation* impose que les seules manipulations licites soient celles de l'interface. En d'autres termes, il n'est possible de manipuler un objet que par le biais des méthodes qui lui sont applicables.
L'ensemble M des signature de méthodes sera associé à la hiérarchie de classes $(C, \sigma, \prec)$. Une signature, peut être vue comme le fait de donner une "sémantique" de m pour c (au moins au niveau syntaxique). Les restrictions suivantes sont imposées sur les signatures des méthodes :

1. *Non ambiguïté* : Si c est une sous-classe de c' et c'' et s'il existe une définition de m pour c' et c'' alors il doit exister une définition de m pour une sous-classe de c' et c'' qui soit c elle-même ou une super-classe de c.

2. *Covariance* : Si $m : c \times \tau_1 \times \ldots \times \tau_n \rightarrow \tau$ et $m : c' \times \tau'_1 \times \ldots \times \tau'_m \rightarrow \tau'$ sont deux définitions distinctes de m et si $c \prec c'$, alors $n = m$ et pour tout i, $\tau_i \leq \tau'_i$ et $\tau \leq \tau'$.

La première règle permet d'éviter d'avoir des ambiguïtés dans le cas où deux implémentations seraient applicables pour un même objet. Pour la deuxième règle, il est nécessaire de faire un peu d'histoire. Habituellement, dans le domaine de la théorie des types, la règle dite de contravariance est utilisée (C.F., cours de module M_4). Cependant, cette règle, parfaitement adaptée au cas du sous-typage des fonctions, ne l'est pas dans le cas des méthodes dans lequel on *masque* les implémentations (on ne les sous-type pas). Pour une synthèse sur le sujet, nous recommandons vivement [Cas95a]. La règle de covariance a été utilisée non pas pour des motifs théoriques (à l'époque les idées n'étaient pas si claires) mais pour des raisons pragmatiques : avec la règle de contravariance, il n'était pas possible d'exprimer des méthodes binaires (comme l'égalité) sans contredire cette règle. La règle de covariance fut alors utilisée bien qu'elle ne garantissait pas un typage statique. De nombreux systèmes ou langages orientés-objet commerciaux ont adopté la règle de covariance. Un moyen de garantir quand même la propriété de typage statique pour ces langages ou systèmes est décrit dans [Cas95b, BC96].

Nous terminons cette présentation avec la définition de la notion de schéma et d'instances.

2.2.4 Schémas et instances

Un schéma, nous l'avons vu précédemment, décrit la structure des données stockées dans la base. Ceci inclut la définition des types, des classes, de hiérarchies d'héritage de classes ainsi que la définition des signatures des méthodes (cad, les interfaces offertes pour les objets de ces classes).Dans les modèles orientés-objet, il faut, en outre, bien isoler le rôle que jouent les identificateurs d'objets (oids). Comme nous le verrons, les oid ont un rôle d'étiquettes ou de variables et donc pour une instance donnée, ne jouent pas un rôle significatif.

De nombreux systèmes commerciaux offrent la possibilité de stocker des valeurs structurées auxquelles il est possible d'avoir accès directement (par leur nom). Un tel mécanisme permet de faire cohabiter harmonieusement les valeurs structurées, les relations avec les classes. En outre, ce mécanisme facilite l'écriture des requêtes. De plus, ce mécanisme est le seul permettant d'assurer la persistance des données de manière transparente à l'utilisateur.

Nous incluons, dans le modèle présenté ici, un tel mécanisme. Un schéma pourra donc comprendre un ensemble de valeurs nommées. Les méthodes, le langage de requêtes et plus généralement, les langages de programmation associés pourront référencer de telles valeurs nommées. Dans le modèle présenté, les classes et les valeurs nommées sont des valeurs dénotables du langage.

Pour l'exemple 3, `Pariscope, PersonnesAimees, ActeursPreferes` et `RealisateurPrefere` sont des valeurs nommées. `Pariscope` référence un ensemble de triplets (une relation), `PersonnesAimees, ActeursPreferes` un ensemble d'objets `Personne`, (resp., `Acteurs`) et `RealisateurPrefere` un unique objet de la classe `Realisateur`.

Definition 14 Un schéma est un quintuple **S** = $(C, \sigma, \prec, M, G)$ où

1. G est un ensemble de *noms* disjoint de C,
2. σ est une fonction de $C \cup G$ vers **types**(C),
3. $(C, \sigma, \prec)$ est une hiérarchies bien formée et,
4. M est un ensemble bien formé de signatures de méthodes.

Une instance d'un schéma peuple les classes avec des oids, assigne des valeurs à ces oids, donne un sens aux racines de persistance et donne une sémantique aux signatures des méthodes. La sémantique des signatures de méthodes est un ensemble de fonctions décrivant leurs effets. D'un point de vue pratique, les classes, les valeurs des objets et les valeurs des noms sont définies de façon *extensionnelle* alors que la sémantique des méthodes est décrite au moyen de "code" et est définie de manière *intensionnelle*. Nous ne considérons pas, dans ce qui suit, le code des méthodes (leur sémantique).

Definition 15 Une *instance* du schéma $(C, \sigma, \prec, M, G)$ est un quadruple **I** = (π, ν, γ, μ) où

1. π est une assignation d'oid (et $O = \cup\{\pi(c); c \in C\}$),
2. ν est une fonction de O vers **val**(O) préservant le typage, c'est à dire, pour tous c et $o \in \pi(c), \nu(o) \in dom(\sigma(c))$,
3. γ est une fonction associant à chaque nom de G de type τ, une valeur dans $dom(\tau)$,
4. μ associe à chaque méthode de M, sa sémantique. Plus précisément, pour chaque signature $m : c \times \vec{\alpha} \to \tau$,

$$\mu(m : c \times \vec{\alpha} \to \tau) : dom(c \times \vec{\alpha}) \to dom(\tau)$$

   Soit donc, une fonction partielle de $dom(c \times \vec{\alpha}) \to dom(\tau)$

La fonction $\mu(m : c \times \vec{\alpha} \to \tau)$ n'a de sens que pour les objets de c et des sous-classes de c pour lesquelles m n'a pas été redéfinie.
Intuitivement, la sémantique des domaines atomiques est bien entendue. En revanche, les oids utilisés dans une instance donnée ne sont pas significatifs, la seule information véritable consiste en les liens que peuvent avoir entre-eux deux oids ou avec des constantes. D'un point de vue pratique, ceci a deux retombées. D'une part, dans la plupart des langages, il n'est pas possible de manipuler explicitement les oids. D'autre part, au niveau physique, un oid peut être amené à changer au cours de la vie de l'objet auquel il correspond (suite à un mécanisme de ramasse miettes ou de regroupement).

Pour prendre en compte cet aspect, dans le modèles formel, nous introduisons la notion d'isomorphisme d'oids. Deux instances **I** et **J** sont *oid-isomorphes* , noté $\mathbf{I} \equiv_{oid} \mathbf{J}$, s'il existe une bijection sur **dom** ∪ **obj** qui soit l'identité sur **dom** et telle que l'image de **obj** soit **obj** et qui transforme **I** en **J**. Pour être précis, une instance d'un schéma orienté-objet devrait se référer à une classe d'équivalence modulo isomorphisme d'oid. Cependant, il est plus commode de travailler avec un représentant.

Pour résumer, dans le modèle proposé, une classe regroupe deux aspects :

1. au niveau du schéma, la définition de classe (type, signatures des méthodes) et
2. au niveau instance, l'extension de la classe (l'ensemble des objets véritablement dans la classe).

Ceci a une incidence non négligeable, en particulier, pour détruire un objet de la base, il faut le détruire *explicitement* de la classe à laquelle il appartient. Dans certains cas, ceci n'est pas pratique. Supposons par exemple que nous ayons une base contenant une classe `Polygone` et supposons que les polygones soient seulement référencés par des figures. Lorsqu'un polygone donné n'est plus utilisé, il n'a plus d'intérêt et devrait être détruit. Une telle destruction devrait être *implicite* (sinon l'utilisateur doit chercher partout où un tel polygone peut apparaïtre dans la base). Ce mécanisme correspond à la notion de persistance par atteignabilité. On défini un ensemble de noms (les racines de persistance) et tout ce qui est atteignable à partir de ces noms persiste, ce qui n'est plus atteignable est automatiquement détruit.

Chapter 3

Modèles Semi-structurés

Voir transparents.

Bibliography

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Database Systems*, chapter 21, pages 542–578. Adison Wesley, 1995.

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *ACM SIGMOD International Conference*, June 1989. to appear in JACM.

[BC96] John Boyland and Giuseppe Castagna. Type-safe compiling of covariant specialization: a practical case. Unpublished manuscript. Currently available by anonymous ftp from `ftp.ens.fr` in file `/pub/dmi/users/castagna/o2.ps.Z`, 1996.

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[Cas95b] G. Castagna. A proposal for making O_2 more type safe. Rapport de Recherche liens-95-4, LIENS, March 1995.

[LR89a] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *ACM PODS International Conference*, March 1989.

[LR89b] C. Lécluse and P. Richard. The O_2 Database Programming Language. In *VLDB International Conference*, August 1989.

[LRV88] C. Lécluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *ACM SIGMOD International Conference*, May 1988.