Notes du cours 1

1 Présentation du cours

Objectifs :*Ce cours présente les bases de la programmation objet et de la programmation concurrente et distribuée puis forme à la programmation d'applications distribuées sur Internet. Exposé du plan du cours. Brève description du langage Caml.*

1.1 Equipe pédagogique, plan du cours et bibliographie

1.1.1 Equipe pédagogique

- Emmanuel Chailloux , Jussieu : couloir 46-00 2 ème porte 211, `Emmanuel.Chailloux@lip6.fr`
- Pascal Manoury , Jussieu : couloir 46-00 2 ème porte 211, `Pascal.Manoury@lip6.fr`
- Bruno Pagano , CEA : C475 (8 rue Captain Scott), `Bruno.Pagano@lip6.fr`

1.1.2 Plan du cours

Ce plan est donné à titre indicatif. Il est fort probable que de nombreux changements auront lieu en cours d'année. Ce cours tient compte du planning du cours "Programmation" obligatoire de la maîtrise, en particulier pour sa partie programmation objet en Java qui débutera début novembre. Ce cours s'articule sur trois axes : la programmation Objet (O'CAML et Java, typage, comparaison avec la programmation modulaire), la programmation parallèle (modèles à mémoire partagée et à mémoire distribuée) et sur la distribution d'applications (code mobile, sécurité et représentation).

1. this ou self
2. Modules paramétrés en Objective Caml (O'Caml), étude du module alpha-beta, application à la programmation de jeux à deux joueurs et à la réalisation de script CGI.
3. Objets en O'Caml : classes et instances, héritage, classe abstraite.
4. Objets en O'Caml : héritage multiple, inférence de types et typage des objets.
5. Processus légers (threads) : modèle de programmation parallèle avec mémoire partagée, concurrence, section critique, exclusion mutuelle, sémaphore.
6. Modèle client/serveur : en utilisant la bibliothèque Unix d'O'CAML (sockets, select,...)
7. Présentation du langage Java : comparaison avec O'Caml.
8. Construction d'applets (petites applications graphiques), bibliothèques Java, programmation graphiques.
9. Application distribuée (programmation parallèle avec mémoire distribuée) en Java , modèle client/serveur, distribution de données, distribution d'objets (CORBA).
10. Distribution d'applications, code mobile, accès URL.
11. Sécurité du code mobile, typage à la O'Caml, vérification de types au chargement du code, étude et comparaison des navigateurs NetScape et MMM.
12. programmation graphique en 2D et 3D : langage de la réalité virtuelle (VRML), avec VrcaML, Java3D et Scol .

Ce cours est accessible à l'adresse suivante : http://www-spi.lip6.fr/ emmanuel/Public/enseignement/pod_98_index.html Il gagne à être consulté via un navigateur pour pouvoir suivre les lines indiqués.

1.1.3 Bibliographie

La bibliographie donnée ici est incomplète. Elle augmentera au fur et à mesure de l'avancement des cours. Les deux premiers ouvrages seront utilisés pendant tout le semestre.

- "Processus Concurrents", Ben Ari, Masson 1986.
- "Concurrent Programming : Principles and practices", G. Andrews, Benjamin Cumming, 1991;
- "Java : de l'esprit à la méthode - distribution d'applications sur Internet", M. Bonjour, G. Falquet, J. Guyot et A. Le Grand - International Thomson Publisher, 1996.

Les documentations des logiciels du domaine public utilisés se trouvent sur le serveur des étudiants.

- Documentations en ligne des langages O'Caml et Java.

Pour une mise à niveau sur le langage Caml :

- "Approche Fonctionnelle de la Programmation", G. Cousineau et M. Mauny, EdiSciences, 1995.

Enfin les deux derniers ouvrages comportent de bonnes présentations du langage HTML. D'autres références sur HTML suivent.

- "UNGI 97 : Un Nouveau Guide Internet", Gilles Maire, Edition First, 1997.
- " HTML et la programmation de serveurs Web", Philippe Chaléat et Daniel Charnay, Eyrolles, 1996.

1.1.4 Evaluation

L'évaluation consistera en une partie projet (40%) et une partie écrit (60%). La partie projet comprendra un projet en O'Caml sur un problème de concurrence et de client/serveur et un projet Java sur un problème distribué. Chaque projet comptera pour 20% de la note finale. La partie écrit sera un examen individuel sur table.

1.1.5 Horaires et salles

- cours : le lundi de 16h00 à 18h00 couloir 45-46 salle 406
- TD1 : le jeudi de 16H00 à 19H00 couloir 44-45 salle RC07
- TD2 : le vendredi de 13h30 à 16h30 couloir 22-32 salle 309

1.2 Concurrence et Distribution

On abordera dans ce cours les deux formes principales de programmation parallèle que sont les modèles à mémoire partagée ou à mémoire distribuée. Dans le premier cas, on imagine aisément les problèmes que posent l'accès concurrent de différents processus dans une zone mémoire commune, en particulier quand il y a modification physique de celle-ci. Dans le deuxième cas, la difficulté provient des communications de valeurs entre les processus. Il y a les risques, soit d'attendre indéfiniment en réception, soit de perdre de l'information. De plus la représentation des valeurs communiquées doit être bien établie sous peine d'incompréhension entre les processus. D'où la nécessité de décrire des protocoles de communication et/ou d'établir des formats d'échange de données.

Enfin, il faut noter que ce type de programmation permet d'exprimer les algorithmes d'une nouvelle manière, apportant un nouveau niveau d'abstraction offrant ainsi une expressivité plus importante.

On utilisera les bibliothèques de processus légers (threads) des langages O'Caml et Java pour illustrer la concurrence, et le réseau Internet pour les concepts de distribution.

Notes du cours 2

2 Les modules d'O'Caml

Objectifs :*Présentation du langage O'CAML, comparaison avec le langage caml-light. Principes de programmation modulaire. Description du système de modules de O'CAML*

2.1 Le langage Caml

Le langage Caml (Categorical Abstract Machine Language) est un langage de la famille ML, i.e. un langage fonctionnel typé muni de traits impératifs. Ses caractéristiques principales sont les suivantes.

Le langage Caml est :

- un langage fonctionnel :

  où les fonctions sont des valeurs du langage et peuvent donc être passées comme arguments ou retournées comme résultat de calculs;
- typé statiquement :

  le type des expressions est vérifié à la compilation, une incohérence de types rejette le programme;
- inférant le type des expressions :

  le programmeur n'a pas besoin d'indiquer le type des expressions y compris les paramètres de fonctions, le système déduira automatiquement leur type le plus général;
- polymorphe paramétrique :

  les types peuvent être paramétrés par des variables de types permettant ainsi la définition d'un traitement générique des structures de données analogues;
- acceptant la définition par cas :

  où la définition d'une fonction s'effectue par filtrage de motif sur les arguments passés;
- muni d'un mécanisme d'exceptions :

  permettant de déclencher une exception à tout moment de l'exécution, celle-ci sera alors traitée par le dernier récupérateur d'exceptions rencontré;
- de traits impératifs :

  comprenant les entrées/sorties, l'affectation et les structures de contrôle séquentielle et itérative;
- et d'un système de modules :

  permettant la réutilisation de composants logiciels;
- comprenant de nombreuses bibliothèques :

  en particulier les bibliothèques graphiques et d'analyses lexicale et syntaxique;
- et acceptant deux modes de compilation :

l'un sous forme de boucle d'interaction permettant le développent incrémental d'applications, et l'autre en mode ligne de commande produisant des programmes exécutables autonomes et s'intégrant dans les outils de développement classiques de dépendance de fichiers et de gestion de versions.

Le langage Caml est développé à l'Inria depuis 1984 . Il est principalement utilisé dans l'enseignement, y compris comme premier langage , et la recherche. L'intérêt de son apprentissage provient principalement de son système de typage et de sa sémantique bien définie. Il existe trois dialectes du langage Caml :

- Caml (V3.1) : l'ancêtre dont l'implantation utilisait la bibliothèque d'exécution du langage Le_Lisp; on lui doit le nom actuel car son implantation était basée sur la machine CAM.
- Caml-Light :
- Objective Caml (O'Caml) :

Caml-Light a été étudiée dans le cours de Programmation et utilisé dans le cours de Compilation de la licence d'informatique de Paris 6. Pour les étudiants ne l'ayant jamais étudié, il est fortement conseillé de lire la première partie du livre de Guy Cousineau et Michel mauny : Approche Fonctionnelle de la Programmation .

O'Caml reprend le noyau fonctionnel et impératif de Caml-Light. Il lui ajoute un système de modules paramétrés, une extension objet (classes, instances, héritage, classes abstraites, classes paramétrées, héritage multiple) qui s'intègre à son système de typage. De plus un compilateur optimisant, produisant du code natif pour les principales architectures machine, permet d'obtenir de bonnes performances à l'exécution. Enfin de nouvelles bibliothèques, principalement la bibliothèque de tt threads (processus légers), permet de l'utiliser comme support d'un cours sur la concurrence.

Nous étudierons dans ce cours son système de modules.

2.2 La programmation modulaire et les modules d"O'CAML

La programmation modulaire permet le découpage d'un programme en *unités logiques* plus petites. Le but étant de pouvoir traiter (réaliser) un module séparément des autres modules, y compris de ceux dont il dépend. Pour cela un module possède une interface qui décrit les communications avec les objets (valeurs, types, exceptions, ...) du module. Enfin au moment de l'assemblage des différents modules les hypothèses des interfaces doivent être vérifiées. L'intérêt est donc :

- découpage logique;
- abstraction des données (spécification et réalisation);
- indépendance de l'implantation;
- réutilisation.

Il y a souvent une confusion entre programmation modulaire et compilation séparée. La compilation séparée décompose un programme en unités de compilation, compilables séparément. Les deux approches sont nécessaires pour la réalisation d'applications de grande taille. Pour ce faire, il est nécessaire que la spécification d'un module (au sens "types abstraits de données) soit vérifiable par un compilateur. Pour cela on se limitera à la vérification de types sans chercher à vérifier d'autres propriétés du module. Pour que les deux approches se rejoignent une interface sera à la fois la spécification d'un module et contiendra l'information de typage et de compilation pour les autres modules.

2.2.1 Le langage de modules d'O'Caml

On parlera de *structure* pour pour la partie réalisation/implantation et de *signature* pour la partie spécification/interface. Le langage de modules est indépendant du langage de base (O'Caml), bien qu'il y ait un parallèle entre *(valeur : type)* et *structure : signature*. On peut voir la signature d'un module comme son type.

modules simples La partie implantation d'un module est une suite de définitions : de valeurs y compris fonctionnelles, de types, d'exceptions et de sous-modules.

```
module Liste =
  struct
    type 'a liste = Vide | Elt of 'a * 'a liste
    exception Liste_vide
    let nil = Vide
    let ajoute x l = Elt (x,l)
    let tete l = match l with Vide -> raise Liste_vide | Elt(t,q) -> t
    let queue l = match l with  Vide -> raise Liste_vide | Elt(t,q) -> q
    let rec longueur l = match l with Vide -> 0 | Elt(_,q) -> 1 + longueur q
  end;;
```

qui donne la signature suivante :

```
module Liste :
  sig
    type 'a liste = Vide | Elt of 'a * 'a liste
    exception Liste_vide
    val nil : 'a liste
    val ajoute : 'a -> 'a liste -> 'a liste
    val tete : 'a liste -> 'a
    val queue : 'a liste -> 'a liste
    val longueur : 'a liste -> int
  end
```

L'accès à un élément d'un module se fait par la notation "point".

```
# Liste.nil;;
- : 'a Liste.liste = Liste.Vide
```

qui peut être simplifié par l'ouverture du module :

```
# open Liste;;
# ajoute "hello" nil;;
- : string Liste.liste = Elt ("hello", Vide)
```

La partie interface d'un module est une suite de déclarations et de spécification de types. On utilisera la convention suivante pour nommer les structures et les signatures : une signature sera écrite en MAJUSCULE, et une structure en Minuscule dont l'initiale en majuscule.

```
module type LISTE =
  sig
    type 'a liste = Vide | Elt of 'a * 'a liste
    exception Liste_vide
    val nil : 'a liste
    val ajoute : 'a -> 'a liste -> 'a liste
    val tete : 'a liste -> 'a
    val queue : 'a liste -> 'a liste
  end
```

Quand une signature est associée à une structure il y a vérification de la cohérence :

- les déclaration de la signature existent dans la structure
- et satisfont les spécifications de la signature.

communications entre modules Les modules utilisent des déclarations d'autres modules, c'est ce que l'on appelle la communication entre modules. Elles peuvent être implicites, en utilisant la notation "point" et en tenant compte de l'environnement global ou explicite en utilisant des foncteurs.

de manière implicite

```
module Element =
  struct
    type t = int
  end;;

module ListeV2 =
  struct
    type liste = Vide | Elt of Element.t * liste
                 (* on suppose Element connu dans l'environnement global *)
    exception Liste_vide
    let nil = Vide
    let ajoute x l = Elt (x,l)
    let tete l = match l with Vide -> raise Liste_vide | Elt(t,q) -> t
    let queue l = match l with  Vide -> raise Liste_vide | Elt(t,q) -> q
    let rec longueur l = match l with Vide -> 0 | Elt(_,q) -> 1 + longueur q
  end;;
```

de manière explicite : module paramétré (foncteur)

```
module type ELEMENT =
  sig
    type t
  end;;

module ListeV3 = functor (Element : ELEMENT) ->
  struct
    type liste = Vide | Elt of Element.t * liste
    exception Liste_vide
    let nil = Vide
    let ajoute x l = Elt (x,l)
    let tete l = match l with Vide -> raise Liste_vide | Elt(t,q) -> t
    let queue l = match l with  Vide -> raise Liste_vide | Elt(t,q) -> q
    let rec longueur l = match l with Vide -> 0 | Elt(_,q) -> 1 + longueur q
  end;;
```

et de son application :

```
module ListeV4 = ListeV3(Element);;

module NouvelElement =
  struct
    type t = float
  end;;

module ListeV5 = ListeV3(NouvelElement);;
```

syntaxe des modules La déclaration d'une structure est "typecheckée", ces informations de typage produisent une signature. La coercion d'une structure par une signature utilise la syntaxe de la coercion des types. En cela le langage de module est un langage fonctionnel typé.

- valeur :

```
module Nom [ : SIGNATURE ] =
  struct
      val ...
      type ...
      exception ...
      module ...
  end
```

- type

  ```
  module type Nom =
    sig
      ...
    end
  ```

- abstraction (valeur fonctionnelle)

  ```
  module Nom = functor ( Module : SIGNATURE) ->
    struct
    end
  ```

- application

  ```
  module Nom = Module(Structure)
  ```

Abstraction Les déclarations de type dans les signatures peuvent être concrètes (la définition du type est alors visible) ou abstraite (la représentation du type n'est pas visible). Ce dernier cas permet de faire de construire des types abstraits de données (ADT).

abstraction de types

```
module type ELEMENT_ORD =
  sig
    type t
    val compare : t -> t -> bool
  end
;;

module type ARBRE_BIN  = functor (Element : ELEMENT_ORD) ->
  sig
     type element = Element.t
     type arbre_bin
     val arbre_vide : arbre_bin
     val ajoute : element -> arbre_bin -> arbre_bin
     val appartient : element -> arbre_bin -> bool
  end;;

module ArbreBin = functor (Element : ELEMENT_ORD) ->
  struct
    type element = Element.t
    type arbre_bin = Vide | Noeud of element * arbre_bin * arbre_bin
    let arbre_vide = Vide
    let rec ajoute x a =
      match  a with
        Vide -> Noeud (x,Vide,Vide)
      | Noeud(l,g,d) ->
          if Element.compare x l
          then Noeud(l,ajoute x g,d)
          else Noeud(l,g,ajoute x d)
    let rec appartient x a =
      match a with
        Vide -> false
      | Noeud(l,g,d) ->
          if x = l then true
          else
            if Element.compare x l
            then appartient x g
            else appartient x d
```

```
  end;;

module ArbreBinAbstrait = (ArbreBin : ARBRE_BIN);;

module EntierInf  =
  struct
    type t = int
    let compare x y = x < y
  end;;


module ArbreBinInf  = ArbreBinAbstrait(EntierInf);;
```

L'utilisation du module `ArbreBinInf` est la suivante :

```
#   open ArbreBinInf;;
# let a = arbre_vide;;
val a : ArbreBinInf.arbre_bin = <abstr>
# ajoute 3 a;;
- : ArbreBinInf.arbre_bin = <abstr>
# ajoute "hello" a;;
This expression has type string but is here used with type
  ArbreBinInf.element = int
```

L'intérêt de l'abstraction dans les modules par rapport au polymorphisme paramétrique de ML, est d'effectuer une limitation du polymorphisme (dans un but de vérification). En effet si l'on définit un nouveau module `ArbreBinSup` représentant les arbres de recherche sur les entiers ordonnées par *supérieur*, dont la représentation interne des arbres est identique, il ne sera en aucun possible d'appliquer `ArbreBinInf.ajoute` sur un arbre d'`ArbreBinSup`. Dans le cas d'une fonction polymorphe d'ajout, prenant comme premier élément l'ordre, il est possible de casser la structure d'un arbre de recherche si on l'applique avec un autre ordre.

contraintes de partage

Si pour s'abstraire, encore un peu plus, on désire déclarer abstraitement le type `element`, il sera alors nécessaire d'indiquer un partage de types.

```
module EntierInfAbstrait = (EntierInf : ELEMENT_ORD);;

module NouvelArbreBinInf =
  ArbreBinAbstrait(EntierInfAbstrait);;
```

la même séquence d'utilisation que précédemment provoquera une erreur de typage :

```
# open NouvelArbreBinInf;;
# let a = arbre_vide;;
val a : NouvelArbreBinInf.arbre_bin = <abstr>
# ajoute 3 a;;
This expression has type int but is here used with type
  NouvelArbreBinInf.element = EntierInfAbstrait.t
```

L'égalité des types `NouvelArbreBinInf.element` et `ElementInfAbstrait.t` est perdue. Pour la rétablir il est nécessaire de l'indiquer explicitement.

En fait il est même nécessaire d'ouvrir la signature `ARBRE_BIN` pour pouvoir effectuer cette contrainte de types. Comme `ARBRE_BIN` est un foncteur, il n'est pas possible de l'appliquer dans le but d'obtenir la signature résultat. On définit alors une nouvelle signature `X_ARBRE_BIN` de la manière suivante :

```
module type X_ARBRE_BIN =
  sig
    type element
    type arbre_bin
    val arbre_vide : arbre_bin
    val ajoute : element -> arbre_bin -> arbre_bin
    val appartient : element -> arbre_bin -> bool
  end;;
```

où le type `element` est abstrait. Le foncteur suivant rétablit la correspondance des types.

```
module XArbreBinAbstrait  =
 (ArbreBinAbstrait :
   functor (E : ELEMENT_ORD) -> (X_ARBRE_BIN with type element = E.t));;
```

On retrouve dans la signature du foncteur `XArbreBinABstrait` l'information suivante : `type element = E.t`.

```
module XArbreBinInf = XArbreBinAbstrait(EntierInf);;
```

Et l'on se retrouve dans le cas où le type `element` est lié au type `t`, tout en l'ayant déclaré abstrait dans la signature `X_ARBRE_BIN`.

```
# open XArbreBinInf;;
# let a = arbre_vide;;
val a : XArbreBinInf.arbre_bin = <abstr>
# ajoute 3 a;;
- : XArbreBinInf.arbre_bin = <abstr>
```

Compilation séparée L'unité de compilation en O'Caml reste le fichier, à la manière de Caml-Light, où une unité de compilation est découpée en deux fichiers: un fichier d'interface d'extension `.mli` pour la signature et un fichier d'implantation d'extension `.ml` pour la structure. Si l'on ne précise rien, le module correspondant à l'unité de compilation est :

```
module Nom = (
   struct
     contenu du fichier nom.ml
   end :
   sig
     contenu du fichier nom.mli
   end
)
```

Attention le nom du module qui commence par une majuscule correspond aux fichiers `nom.ml` et `nom.mli` qui commencent par une minuscule.

L'environnement de typage est celui des répertoires d'accès aux fichiers, c'est à dire le répertoire courant, la bibliothèque standard, ...

2.3 De Caml-Light à O'Caml

Pour porter un programme Caml-Light en O'Caml, il est nécessaire de :

- réécrire les ouvertures de fichier : on remplacera alors le `#open "fichier";;` par un `open Fichier;;`;
- de renommer certaines fonctions des bibliothèques standards.

Dans le cas de la boucle d'interaction, le `#use "fichier.ml";;` remplacera le `load "fichier";;`.

2.4 Autres lectures

Ce cours s'inspire des lectures suivantes :

- La documentation d'O'Caml sur les modules ;
- le cours de Xavier Leroy au DEA SPP, en PostScript ;
- le rapport Inria 2721 sur la version intermédiaire de Caml, nommée Caml Special Light, en PostScript gzippé ;
- la présentation de Maria-Virginia Aponte à l'école de jeunes chercheurs du GDR de Programmation de 1995.

elles ne sont pas obligatoire mais permettent d'avoir une vision plus complète de la programmation modulaire et des modules d'O'Caml.

Notes du cours 3

3 Les objets d'O'Caml : partie I

Objectifs :*Présentation du noyau objet du langage O'CAML : classes, héritage, classes et méthodes abstraites*

3.1 Classes et Objets

L'extension objet d'O'Caml s'intègre au noyau fonctionnel et au noyau impératif du langage, et aussi à son système de types. C'est ce dernier point qui en fait son originalité. On obtient ainsi un langage objet, typé statiquement, avec inférence de types. Cette extension permet de définir des classes et des instances, autorise l'héritage entre classes y compris multiple, accepte les classes paramétrées et les classes abstraites. Les interfaces de classes sont engendrées par leur définition mais peuvent être précisées par une signature de modules.

3.1.1 classes

On appelle classe la description du regroupement des données et des procédures (méthodes) qui les manipulent.
On définit l'inévitable classe `point` qui contient deux champs de données (ou variables d'instance), et six champs de méthodes (ou méthodes d'instance) : deux méthodes d'accès aux champs de données, deux procédures de déplacement absolu et relatif d'un point, un affichage et une fonction de calcul de distance par rapport à l'origine. Cette définition de la classe `point` possède des méthodes effectuant des modifications physiques des champs de données.

```
class point (x_init,y_init) =
object
  val mutable x = x_init
  val mutable y = y_init

  method get_x = x
  method get_y = y
  method moveto (a,b) = begin x <- a; y <- b end
  method rmoveto (dx,dy) = begin x <- x + dx; y <- y + dy end
  method print () =
    begin
      print_string "( ";
      print_int x;
      print_string " , ";
      print_int y;
      print_string ")";
     end

  method distance () = sqrt(float(x*x + y*y))

end;;
```

Le système infère l'interface de la classe :

```
class point :
  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
  end
```

3.1.2 instances

Un objet est une valeur d'une classe, appelée *instance* de cette classe. Cette instance est créée par le constructeur d'objets `new` à qui on indique la classe et les valeurs d'initialisation.

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
```

Le type inféré pour les instances `p1` et `p2` est le type objet (`<obj> point`). C'est une abréviation du type objet long suivant :

```
  point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; print : unit -> unit;
      rmoveto : int * int -> unit >
```

contenant les méthodes et leur type.

3.1.3 envoi de message

L'envoi d'un message à un objet s'effectue par la notation `#` (la notation point étant déjà utilisée pour les records et les modules). Ce message correspond à une méthode définie dans la classe de l'objet. L'exemple suivant montre différentes requêtes effectuées sur des objets de la classe `point`.

```
# p1#get_x;;
- : int = 0
# p2#get_y;;
- : int = 4
# p1#affiche();;
( 0 , 0)- : unit = ()
# p2#print();;
( 3 , 4)- : unit = ()
# if (p1#distance()) = (p2#distance())
then print_string ("c'est le hasard\n")
else print_string ("on pouvait parier\n");;
on pouvait parier
- : unit = ()
```

Du point de vue des types, les objets de type `point` peuvent être manipulés par les fonctions polymorphes d'O'Caml :

```
# p1 = p1 ;;
- : bool = true
# p1 == p1;;
- : bool = true
# p1 = p2;;
- : bool = false
# let p3 = new point (0,0);;
val p3 : point = <obj>
# p1 = p3;;
- : bool = true
# p1 == p3;;
- : bool = false
```

comme n'importe quelle valeur du langage.

3.2 Agrégat

Une classe peut contenir des champs de données (ou variables d'instance) appartenant à d'autres classes. C'est la relation "Has-a" (ou "A-un") entre deux classes. Elle est notée par une simple flèche entre les 2 classes dans le sens "C1 Has-a C2" : `C1 --> C2`. Si `C1` a de 0 à n champs de `C2` on notera : `C1 <>---> C2` la relation.
L'exemple suivant définit une classe `picture` contenant un tableau de `point`

```
class picture n =
object
  val mutable ind = 0
  val tab = Array.create n (new point(0,0))

  method add p = if (ind < n -1) then
                 begin
                   tab.(ind)<-p;
                   ind <- ind + 1
                 end
                 else failwith ("picture.add : ind = "^(string_of_int ind))

  method remove () = if (ind > 0) then ind <-ind-1

  method print () = for i=0 to ind do tab.(i)#print() done
end;;
```

Le système infère l'interface de la classe :

```
class picture :
int ->
object
  val mutable ind : int
  val tab : point array
  method add : point -> unit
  method print : unit -> unit
  method remove : unit -> unit
end
```

Le champs `tab` possède le type `point array` correspondant à un tableau de points.

3.3 Héritage

C'est l'avantage majeur de la programmation objet que de pouvoir étendre le comportement d'une classe existante tout en continuant à utiliser le code écrit par la classe originale. Quand on étend une classe, la nouvelle classe hérite de tous les champs, de données et de méthodes, de la classe qu'elle étend.

C'est la relation "Is-a" (ou "Est-un") entre 2 classes. Elle est notée par une flèche remplie entre la sous-classe et la classe ancêtre.

3.3.1 héritage d'une classe

Voici une extension de la classe `point` qui hérite des coordonnées et des déplacements de point et du calcul de distance :

```
class point_colore p c =
object
  inherit point p

  val c = c

  method get_color = c
  method print () =
  begin
      print_string "( ";
      print_int x;
      print_string " , ";
      print_int y;
      print_string (") de couleur "^c);
  end
end;;
```

Le système retourne l'interface de classe suivante :

```
class point_colore :
  int * int ->
  string ->
  object
    val c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_color : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
  end
```

Toutes les méthodes de l'interface peuvent être utilisée :

```
# let pc = new point_colore (2,3) "blanc";;
val pc : point_colore = <obj>
# pc#get_color;;
- : string = "blanc"
# pc#get_x;;
- : int = 2
# pc#display();;
( 2 , 3) de couleur blanc- : unit = ()
# pc#get_x;;
- : int = 2
```

La classe point_colore redéfinit la méthode print pour tenir compte du champ couleur. On dit que la méthode print redéfinit celle de son ancêtre. Elle doit avoir le même type. Cela ne suffit pas pour rendre les types point et point_colore compatibles :

```
# p1 = pc;;
This expression has type
  point_colore =
    < distance : unit -> float; get_color : string; get_x : int; get_y :
      int; moveto : int * int -> unit; print : unit -> unit;
      rmoveto : int * int -> unit >
but is here used with type
  point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; print : unit -> unit;
      rmoveto : int * int -> unit >
Only the first object type has a method get_color
```

3.3.2 référencement : self et super

Il est pratique, dans la définition d'une méthode d'une classe, de pouvoir invoquer une autre méthode de la classe sur soi-même ou de pouvoir invoquer une méthode de la classe ancêtre. Pour cela O'Caml autorise de nommer l'objet soi-même ou la classe ancêtre. On redéfinit la classe point_colore :

```
class point_colore p c  =
object(self)
  inherit point p as super

  val c = c

  method get_color = c
  method print () =
  begin
      super#print();
      print_string (" de couleur "^ self#get_color);
  end
end;;
```

Il est possible de donner des noms quelconques pour la classe ancêtre et sa sous-classe, mais autant utiliser la terminologie objet (self ou this pour soi-même et super pour l'ancêtre). Cela est utile dans le cas de l'héritage multiple pour différencier les ancêtres.

3.3.3 liaison retardée

On appelle "liaison retardée" (ou liaisons "dynamique") la détermination à l'exécution de la méthode à utiliser lors de l'envoi d'un message. La liaison "précoce" (ou liaison "statique") effectue cette résolution à la compilation
Quand un programme s'exécute, il doit établir la valeur associée à chaque identificateur rencontré. Cette liaison entre un identificateur et sa valeur peut s'effectuer soit à la compilation (liaison statique ou précoce), soit à l'exécution (liaison dynamique ou retardée). Ce problème se posait avant la programmation par objet (par exemple la majorité des dialectes Lisp interprétés possèdent une liaison dynamique, et les dialectes ML compilés une liaison statique).
En programmation objet les liaisons concernent aussi les méthodes. Les langages à objets utilisent la liaison retardée pour implanter le polymorphisme *ad hoc* où le même envoi de message peut déclencher différents codes à s'exécuter selon l'objet receveur. C'est l'objet lui-même qui saura le code à exécuter. On appelle

surcharge d'une méthode le fait de conserver plusieurs liaisons pour cette méthode (à ne pas confondre avec la redéfinition qui masque une ancienne définition qui n'est plus accessible).

L'exemple de la classe abstraite `expr_ar` illustre ce propos. La fonction `evaluateur` de type `#expr_ar -> unit` prend une expression arithmétique et lui envoie le message `eval`. C'est l'objet lui-même qui pourra déterminer la méthode à employer selon sa nature (constante ou opération binaire). Il est presque impossible au compilateur de le déterminer. En effet le type de la fonction étant le type d'une classe abstraite sans le corps des méthodes (donc sans code) la liaison ne peut qu'être retardée. Cette détermination pourrait être effectuées pour les classes concrètes, mais limiterait l'intérêt du sous-typage.
On suppose que l'on avait écrit la méthode `rmoveto` de la classe `point` en appelant la méthode `get_y` sur `self`. Dans la classe `point_colore` on redéfinit seulement la méthode `get_y` qui retourne la valeur du champs `y` fois 100. C'est un cas d'école, mais cela permet de comprendre le mécanisme.

```
class point (x_init,y_init) =
object(self)
...
  method rmoveto (dx,dy) = begin x <- x + dx; y <- self#get_y + dy end
...
end;;

class point_colore p c =
object
  inherit point p

  val c = c
  method get_y = y*100
  method get_color = c
...
end;;
```

Le programme suivant construit un point et un point_colore, les affiche, les déplace et les réaffiche "

```
# let p = new point (1,1);;
val p : point = <obj>
# p#print();;
( 1 , 1)- : unit = ()
# p#rmoveto(3,4);;
- : unit = ()
# p#print();;
( 4 , 5)- : unit = ()
# let pc = new point_colore(1,1) "blanc";;
val pc : point_colore = <obj>
# pc#print();;
( 1 , 1) de couleur blanc- : unit = ()
# pc#rmoveto(3,4);;
- : unit = ()
# pc#print();;
( 4 , 104) de couleur blanc- : unit = ()
```

La méthode `rmoveto` qui n'a pas été redéfinie a son comportement modifiée par la redéfinition de la méthode `get_y`. En effet la liaison, c'est à dire le choix de la méthode `get_y` dans le corps de `rmoveto` n'est pas déterminé à la compilation de la classe `point`. Ce choix est effectué en regardant dans la liste des méthodes d'une instance de la classe `point` ou `point_colore`. Pour une instance de `point` l'envoi du message `rmoveto` déclenchera la méthode `get_y` définie dans `point`. Par contre le même envoi de message sur une instance de `point_colore`, appellera la méthode `rmoveto` héritée de `point` et déclenchera la méthode `get_y` redéfinie dans `point_colore`.

3.3.4 initialisation

Il est possible d'indiquer dans la définition de la classe, une méthode `initializer` déclenchée immédiatement après la construction de l'objet. Cet "initialisateur" peut faire n'importe quel calcul et a accès aux champs de l'instance (car celle-ci vient d'être créée). On reprend l'exemple des classe `point` et `point_colore` en ajoutant à chaque classe un initialisateur.

```
class point (x_init,y_init) =
object
...
initializer print_string "Creation d'un point";
              print_newline(); flush stdout

end;;

class point_colore p c =
object
  inherit point p
...
  initializer print_string "Creation d'un point colore";
              print_newline(); flush stdout

end;;
```

L'exécution suivante permet de suivre l'ordre de déclenchement de la construction des objets et de leur initialisateur :

```
# let p = new point;;
val p : int * int -> point = <fun>
# let p = new point (3,4);;
Creation d'un point
val p : point = <obj>
# let pc = new point_colore (3,4) "blanc";;
Creation d'un point
Creation d'un point colore
val pc : point_colore = <obj>
```

3.3.5 méthodes privées

Une méthode peut être déclarée `private`. Elle n'apparaîtra pas dans l'interface de la classe et donc dans le type de l'objet. Par contre les méthodes privées sont héritées et pourront donc être utilisée dans la hiérarchie. L'exemple suivant reprend la déclaration de la classe `point` en rendant `private` la méthode `rmoveto` :

```
class point (x_init,y_init) =
...
  method private rmoveto (dx,dy) = begin x <- x + dx; y <- self#get_y + dy end
  method step1 = self#rmoveto(1,1)
...
end;;
```

L'interface ne contient donc pas la méthode `rmoveto` comme le montre l'exemple suivant :

```
# let p = new point (2,3);;
Creation d'un point
val p : point = <obj>
# p#print();;
( 2 , 3)- : unit = ()
# p#step1;;
- : unit = ()
```

```
# p#print();;
( 3 , 4)- : unit = ()
# p#rmoveto(1,1);;
This expression has type point
It has no method rmoveto
```

3.4 Classes et méthodes abstraites

Les classes abstraites sont des classes dont certaines méthodes sont déclarées mais ne possèdent pas de corps. Ces méthodes sont dites alors abstraites. Il n'est pas possible d'instancier une classe abstraite (`new` est interdit). On utilise le mot clé `virtual` pour le préciser.
Si une sous-classe, d'une classe abstraite, redéfinit toutes les méthodes abstraite de l'ancêtre, alors elle devient concrète, sinon elle reste abstraite.

Dans cet exemple on construit une classe abstraite `graphical_object` qui ne contient qu'une seule méthode abstraite `print`.

```
class virtual graphical_object () =
object
  method virtual print : unit -> unit
end;;
```

L'interface calculée est la suivante :

```
class virtual graphical_object :
  unit -> object method virtual print : unit -> unit end
```

La sous-classe `rectangle` suivante hérite de `graphical_object` et définit de manière concrète toutes les méthodes abstraites de `graphical_object`.

```
class rectangle (p1,p2) =
object
  inherit graphical_object ()
  val mutable llc = (p1 : point)
  val mutable ruc = (p2 : point)
  method print () = begin
   print_string "(";p1#print();
   print_string ",";p2#print();
   print_string ")"
  end

end;;
```

Le système retourne l'interface de classe suivante :

```
class rectangle :
  point * point ->
  object
    val mutable llc : point
    val mutable ruc : point
    method print : unit -> unit
  end
```

3.5 Autres lectures

Ce cours s'inspire des documents suivants :

- Le cours de Francois Barthélémy et maria-Virginia Aponte au CNAM.
- La documentation d'O'Caml sur les objets;
- Une courte présentation des objets par Didier Remy

Notes du cours 4

4 Les objets d'O'Caml : partie II

Objectifs :*Présentation du noyau objet du langage O'CAML : héritage multiple, classe paramétréees, contraintes de typage*

4.1 Objet et type

Le type d'un objet est le type de ses méthodes. Par exemple le type `point` est une abréviation du type :

```
point =
  < distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; print : unit -> unit;
    rmoveto : int * int -> unit >
```

C'est un type objet fermé, c'est à dire que le type de toutes ses méthodes est déterminé. Et il n'y en a pas d'autres.

Lors d'un envoi de message l'inférence de types construit un type objet ouvert :

```
# let f x = x#get_x;;
val f : < get_x : 'a; .. > -> 'a = <fun>
# let p = new point(2,3);;
val p : point = <obj>
# f p;;
- : int = 2
```

En effet `f` peut s'appliquer à n'importe quelle instance de classe ayant une méthode `get_x` qui retourne une valeur d'un type quelconque. Un type objet ouvert est représenté par la notation `< ..>`, o
'u les 2 points indiquent la possibilité de l'augmenter par le type de nouvelles méthodes.

Si l'on veut passer de la représentation d'un type objet fermé en un type objet ouvert, on utilisera alors la notation `#type_obj` comme dans l'exemple suivant :

```
# let g (x : #point) = x#amess;;
val g :
  < amess : 'a; distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; print : unit -> unit;
    rmoveto : int * int -> unit; .. > ->
  'a = <fun>
```

où la coercion de type avec `#point` force `x` à avoir au moins toutes les méthodes de `point`, et l'envoi du message `amess` ajoute une méthode au type du paramètre `x`.

4.2 Héritage multiple

L'héritage multiple permet d'hériter des champs de données et des méthodes de plusieurs classes. En cas de noms de champs ou de méthodes identiques, seulement la dernière déclaration, dans l'ordre de la déclaration de l'héritage, sera conservée. Les différentes classes héritées n'ont pas forcément de liens d'héritage entre elles.

L'intérêt de l'héritage multiple est d'augmenter la réutilisabilité des classes.

On définit la classe abstraite `geometric_object` qui déclare deux méthodes `compute_area` et `compute_circ` pour le calcul de la surface et du périmètre.

```
class virtual geometric_object () =
object
  method virtual compute_area : unit -> float
  method virtual compute_circ : unit -> float
end;;
```

On redéfinit la classe `rectangle` de la manière suivante :

```
class rectangle (p1,p2) =
object
  inherit graphical_object ()
  inherit geometric_object ()
  val mutable llc = (p1 : point)
  val mutable ruc = (p2 : point)

  method print () = begin
   print_string "(";p1#print();
   print_string ",";p2#print();
   print_string ")"

  method compute_area() = abs(ruc#get_x - llc#get_x) *
                          abs(ruc#get_t - llc#get_y)

  method compute_circ() = (abs(ruc#get_x - llc#get_x) +
                          abs(ruc#get_t - llc#get_y)) * 2
end
;;
```

On obtient le graphe d'héritage suivant :

```
    graphical_object       geometric_object
                \           /
                 \         /
                  \       /
                 rectangle
```

On aurait pu aussi ne pas réécrire la classe rectangle, mais dériver de rectangle de la manière suivante :

```
graphical_object
            \
             \
              \
        rectangle        geometric_object
                \          /
                 \        /
                  \      /
                 n_rectangle
```

Dans ce cas là, seules les méthodes abstraites de la classe abstraite `geometric_object` auraient du être définies dans `n_rectangle`.

Toujours dans la même veine, les développement de la hiérarchie `graphical_object` et `geometric_object` auraient pu être séparés jusqu'au moment il devenait utile d'avoir une classe possèdant les deux comportements :

```
    graphical_object       geometric_object
              |              |
              |              |
              |              |
    rectangle_graph       rectangle_geo
                \           /
                 \         /
```

```
            \       /
            rectangle
```

Si on suppose que chaque classe `rectanglai` déclare des variables d'instances pour les coins d'un rectangle, on se retrouve dans la classe `rectangle` avec 4 points (2 par coin).

```
class rectabgle (p1,p2) =
inherit rectangle_graph (p1,p2) as super_graph
inherit rectangle_geo (p1,p2) as super_geo
end;;
```

Dans le cas où des méthodes de même type existent dans les 2 classes `rectanglei`, alors seule la dernière est visible. Néanmoins, grace au nommage des class ancêtres (`super...`, il est toujours possible d'invoquer une méthode d'une certaine hiérarchie.

4.3 Classes paramétrées

Les classes paramétrées permettent d'utiliser le polymorphisme paramétrique de Caml dans les classes. Une déclaration de classe peut être paramétrée par des variables de types, comme dans une déclaration de types de Caml. Cela offre de nouvelles possibilités de généricité, pour la réutilisation du code, et s'intègre dans le typage à la ML quand l'inférence de type produit des types paramétrés.

4.3.1 exemple

Par exemple si l'on désire construire une classe `pair` pour décrire les paires, une solution nai"ve serait :

```
class pair a b =
object
  val x = a
  val y = b
  method fst = x
  method snd = y
  end;;
```

et produirait l'erreur de typage suivante :

```
# class pair a b =
  object
    val x = a
    val y = b
    method fst = x
    method snd = y y
    end;;
Some type variables are unbound in this type:
  class pair :
    'a ->
    'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
The method fst has type 'a where 'a is unbound
```

Cette erreur de typage indique que la variable de type `'a` du type de `a`, `x` et `fst` n'est pas liée. Pour que le typage reste correct, il est nécessaire de paramétriser la classe `pair` de la manière suivante :

```
class ['a,'b] pair (a:'a) (b:'b) =
  object
    val x = a
    val y = b
    method fst = x
    method snd = y
  end;;
```

qui indique une interface de classe paramétrée par les variables de type `'a` et `'b` :

```
class ['a,'b] pair (a:'a) (b:'b) =
  object
    val x = a
    val y = b
    method fst = x
    method snd = y
  end;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

En reprenant l'exemple sur les piles, il est donc possible de construire une classe paramétrée sur les piles :

```
class ['a] pile ((x:'a),n) =
object(self)
  val mutable ind = 0
  val tab = Array.create n x

  method is_empty () = if ind = 0 then true else false
  method private is_full () = if ind = n+1 then true else false
  method pop() = if self#is_empty() then failwith "pile vide"
               else ind <- ind -1 ; tab.(ind)
  method push y = if self#is_full() then failwith "pile pleine"
                else tab.(ind) <- y; ind <- ind + 1
end;;
```

qui produit l'interface suivante :

```
class ['a] pile :
  'a * int ->
  object
    val mutable ind : int
    val tab : 'a array
    method is_empty : unit -> bool
    method private is_full : unit -> bool
    method pop : unit -> 'a
    method push : 'a -> unit
  end
```

On peut alors créer et manipuler des piles paramétrées de la manière suivante :

```
# let pi = new pile (0.0,10);;
val pi : float pile = <obj>
# pi#push(3.14);;
- : unit = ()
# let ps = new pile ("hello", 20);;
val ps : string pile = <obj>
# ps#push("hello");;
- : unit = ()
# let pp = new pile (new point(0,0),10);;
val pp : point pile = <obj>
# pp#push(new point(4,5));;
- : unit = ()
```

4.3.2 contraintes de typage

Selon l'usage de la valeur du type paramétré, des contraintes de typage peuvent apparaître dans l'interface inférée. On cherche à construire des listes paramétrées sans utiliser de vecteurs. Pour cela on définit une classe abstraite `['a] liste` ainsi que deux sous-classes : `['a] cons` et `['a] nil` de la manière suivante :

```
class virtual ['a] liste () =
object
  method virtual empty : unit -> bool
  method virtual cons  : 'a -> unit
  method virtual head :  'a
  method virtual tail : 'a liste
  method virtual display : unit -> unit
end;;
```

dont l'interface est

```
class virtual ['a] liste :
  unit ->
  object
    method virtual cons : 'a -> unit
    method virtual display : unit -> unit
    method virtual empty : unit -> bool
    method virtual head : 'a
    method virtual tail : 'a liste
  end
```

puis la classe `cons` :

```
class  ['a] cons (v,l)  =
object (self)
  inherit ['a] liste ()
  val mutable car = (v:'a)
  val mutable cdr = (l:'a liste)
  method empty () = false
  method cons x = cdr<-new cons(car,cdr); car<-x
  method head = car
  method tail = cdr
  method display () =
      begin
        car#print();
        print_string "  ::";
        self#tail#display()
    end
end;;
```

dont l'interface tiendra compte de la contrainte de types sur le paramètre de type `'a`, contrainte due à l'envoi de la méthode `print` sur la variable d'instance `car` de type `'a`.

```
class ['a] cons :
  'a * 'a liste ->
  object
    constraint 'a = < print : unit -> 'b; .. >
    val mutable car : 'a
    val mutable cdr : 'a liste
    method cons : 'a -> unit
    method display : unit -> unit
    method empty : unit -> bool
    method head : 'a
    method tail : 'a liste
  end
```

Où une contrainte de type est posée sur la variable de type `'a` qui indique que le type `'a` doit pouvoir s'unifier avec le type de la contrainte. Cette contrainte provient de la méthode `print` envoyée sur le `car` dans la méthode `display`.

La classe `nil` permet juste de construire une fin de liste, utilisée quand crée une instance de `cons`. Pour cela son implantation de la méthode `cons`. Elle n'a pas de contrainte sur le paramètre de type.

```
exception ListeVide;;
class  ['a] nil () =
object (self)
  inherit ['a] liste ()
  val nil = ()
  method empty () = true
  method cons (x:'a) = failwith "bad argument"
  method head = raise ListeVide
  method tail = raise ListeVide
  method display () = print_string "[]"
end;;
```

avec l'interface inférée suivante :

```
  class ['a] nil :
  unit ->
  object
    val nil : unit
    method cons : 'a -> unit
    method display : unit -> unit
    method empty : unit -> bool
    method head : 'a
    method tail : 'a liste
  end
```

On définit par ailleurs une classe `integer` possèdant une méthode `print`. On pourra ainsi construire une liste d'`integer`.

```
class integer i =
object
val v = i
  method get = v
  method print () = print_int v
end;;
```

La construction de liste est la suivante :

```
# let i1 = new integer 1;;
val i1 : integer = <obj>
# let i2 = new integer 2;;
val i2 : integer = <obj>
# let n = new nil ();;
val n : '_a nil = <obj>
# let l = new cons (i1,n);;
val l : integer liste = <obj>
# l#display();;
1  ::[]- : unit = ()
```

4.3.3 contrainte explicite

Cette contrainte sur les variables de type des classes paramétrées peut être explicite.

```
class virtual printable ()  =
object
  method virtual print : unit -> unit
end;;


class  ['a] cons (v,l)  =
object (self)
  inherit ['a] liste ()
```

```
  constraint 'a = #printable
  val mutable car = (v:'a)
  val mutable cdr = (l:'a liste)
  method empty () = false
  method cons x = cdr<-new cons(car,cdr); car<-x
  method head = car
  method tail = cdr
  method display () =
      begin
        car#print();
        print_string "  ::";
        self#tail#display()
    end
end;;
```

4.4 Autres lectures

Ce cours s'inspire principalement des mêmes documents que le cours précédent :

- Le cours de Francois Barthélémy et maria-Virginia Aponte au CNAM.
- La documentation d'O'Caml sur les objets ;
- Une courte présentation des objets par Didier Remy
- Le cours de Didier Remy de Magistère MMFAI
  et de Roberto Di Cosmo au Magistère MMFAI .

Notes du cours 5

5 Les objets d'O'Caml : partie III

Objectifs :*Présentation du noyau objet du langage O'CAML : relation de sous-typage, style fonctionnel, interfaces et modules, simulation d'héritage avec les modules*

5.1 Sous-typage

Le sous-typage est la possibilité pour un objet d'un certain type d'être considéré et utilisé comme un objet d'au autre type.

5.1.1 exemple

Par exemple, en reprenant les définitions des classes `point` et `point_colore`, il est possible d'indiquer, EXPLICITEMENT, qu'une instance de `point_colore` doit être considérée comme `point`. La relation "est un sous-type de" se note `:>`. On note que `point_colore` est un sous-type de `point` de la manière suivante :

```
point_colore :> point
```

Si le membre gauche de la relation est omis, alors c'est le type de la valeur qui sera considéré comme membre gauche.

Soient les déclarations suivantes

```
# let p = new point (4,5);;
val p : point = <obj>
# let pc = new point_colore (4,5) "blanc";;
val pc : point_colore = <obj>
# let np = (pc :> point);;
val np : point = <obj>
# let np2 = (pc : point_colore :> point);;
val np2 : point = <obj>
```

L'invocation de la méthode `print` sur les différents points donnera :

```
#  p#print();;
( 4 , 5)- : unit = ()
#  pc#print();;
( 4 , 5) de couleur blanc- : unit = ()
#  np#print();;
( 4 , 5) de couleur blanc- : unit = ()
```

où l'envoi d'un message `print` sur `np`, valeur considérée de type `point` déclenche la méthode `print` de la classe `point_colore`.

Cela permet par exemple de construire une liste de points contenant en fait des instances de `point_colore` :

```
# let l = [p;np];;
val l : point list = [<obj>; <obj>]
# List.map (fun x -> x#print()) l;;
( 4 , 5)( 4 , 5) de couleur blanc- : unit list = [(); ()]
```

Cela vient de la liaison tardive (choix de la méthode à utiliser à l'exécution). La combinaison de liaison tardive et sous-typage autorise une nouvelle forme de polymorphisme, le polymorphisme d'inclusion.

5.1.2 sous-typage $\neq$ héritage

Le sous-typage est une notion différente de l'héritage. Il y a deux arguments principaux. Le premier est qu'il est possible de coercer un type classe dans un autre type classe sans que le premier corresponde à un descendant du deuxième (on peut faire du sous-typage sans héritage). En effet, on aurait pu définir la classe `point_colore` de manière indépendante de la classe `point` et forcer le type de l'une de ses instances en type classe `point`.

Deuxièmement il est aussi possible d'avoir un héritage de classes sans pouvoir faire du sous-typage entre instances de ces classes. On ajoute une méthode `egalite` à la classe `point` qui prend un argument de type de la classe en entrée. Dans la sous-classe `point_colore` on redéfinit la même méthode dont le type de l'argument est toujours le type de la classe en cours de définition. Si l'on essaie de faire du sous-typage entre deux instances de ces classes, alors il serait possible d'envoyer le message `egalite` à une instance de `point_colore` qui a été coercée en `point` avec un paramètre de type `point`. Dans ce cas la recherche d'un champs `couleur` dans l'argument de type `point` provoquerait une erreur à l'exécution.

```
class virtual egal () =
object(self : 'a)
  method virtual eq : 'a -> bool
end;;

class point (x_init,y_init) =
object(self)
inherit egal ()

  val mutable x = x_init
  val mutable y = y_init

  method eq q = (self#get_x = q#get_x ) && (self#get_y = q#get_y)
  method get_x = x
  method get_y = y
  method moveto (a,b) = begin x <- a; y <- b end
  method rmoveto (dx,dy) = begin x <- x + dx; y <- y + dy end
  method print () =
    begin
      print_string "( ";
      print_int x;
      print_string " , ";
      print_int y;
      print_string ")";
     end

  method distance () = sqrt(float(x*x + y*y))

end;;

class point_colore p c =
object (self)
  inherit point p as super

  val c = c

  method eq q =  (self#get_x = q#get_x )
               && (self#get_y = q#get_y)
               && (self#get_c = q#get_c)
  method get_c = c
```

```
  method print () =
  begin
      super#print();
      print_string (" de couleur "^ self#get_c);
  end
end;;
```

L'exécution du programme suivant entraîne un (long) message d'erreur :

```
# let p = new point (2,3);;
val p : point = <obj>
# let pc = new point_colore (2,3) "blanc";;
val pc : point_colore = <obj>
# let np = (pc :> point);;
This expression cannot be coerced to type
  point =
    < distance : unit -> float; eq : point -> bool; get_x : int; get_y :
      int; moveto : int * int -> unit; print : unit -> unit;
      rmoveto : int * int -> unit >;
it has type
  point_colore =
    < distance : unit -> float; eq : point_colore -> bool; get_c : string;
      get_x : int; get_y : int; moveto : int * int -> unit;
      print : unit -> unit; rmoveto : int * int -> unit >
but is here used with type
  < distance : unit -> float; eq : point -> bool; get_c : string;
    get_x : int; get_y : int; moveto : int * int -> unit;
    print : unit -> unit; rmoveto : int * int -> unit >
Type
  point_colore =
    < distance : unit -> float; eq : point_colore -> bool; get_c : string;
      get_x : int; get_y : int; moveto : int * int -> unit;
      print : unit -> unit; rmoveto : int * int -> unit >
is not compatible with type
  point =
    < distance : unit -> float; eq : point -> bool; get_x : int; get_y :
      int; moveto : int * int -> unit; print : unit -> unit;
      rmoveto : int * int -> unit >
Only the first object type has a method get_c
```

5.1.3 Formalisation

Cette partie est fortement inspirée du cours de Francois Barthélémy et maria-Virginia Aponte au CNAM.

sous-typage entre objets Soient $t =< m_1 : \tau_1; \ldots; m_n : \tau_n >$ et $t' =< m_1 : \sigma_1; \ldots; m_n : \sigma_n; \tau' >$ où τ' est une suite de méthodes,
on dit que t' est un sous-type de t dans C (contexte de typage), noté $t' \leq t$
si $\sigma_i \leq \tau_i$ pour $i \in \{1, \ldots, n\}$.

appel de fonction Si $f : \sigma \rightarrow \tau$dansC, $a : \sigma'$dansC et $\sigma' \leq \sigma$ dansC
alors (fa) est bien typé dans C et a le type τ.

Une fonction f qui attend un argument de type σ peut recevoir sans danger un argument d'un sous-type de σ.

sous-typage de types fonctionnels Si on définit les classes suivantes :

```
class a =
 ...
  method f : t1 -> t2
 ...
end;;

class b =
 ...
  method f : t3 -> t4
 ...
end;;
```

Si on veut montrer que $b \leq a$ alors il faut vérifier $(t_3 \rightarrow t_4) \leq (t_1 \rightarrow t_2)$. Pour distinguer les deux méthodes f on les nomme : f_a et f_b.

Soient $t_1 \rightarrow t_2$ et $t_3 \rightarrow t_4$ deux types fonctionnels, ils sont en relation de sous-typage :
$(t_3 \rightarrow t_4) \leq (t_1 \rightarrow t_2)$
si et seulement si :

$$t_4 \leq t_2 \text{ (co} - \text{variance)} t_1 \leq t_3 \text{ (contra} - \text{variance)}$$

justifications
Soient les 2 fonctions suivantes bien typées :

```
let g (p : t2) = ...
let h ((o:a),(x:t1)) = g(o#f(x));;
```

avec

```
g : t2 -> nt
h : ( a * t1) -> nt
```

1. co-variance : la fonction g attend un argument de type t_2 ou d'un de ses sous-types. Comme cet argument est dans le corps de h résultat de l'envoi du message $f(x)$, il peut être résultat de l'appel de f_b, donc :
$$type\_res(f_b) \leq type\_res(f_a) \Rightarrow t_4 \leq t_2$$
2. contra-variance : En appliquant f à une instance de b (notée o_b on obtient :
$$h(o_b, x) \Rightarrow \mathrm{g}(o_b \# f_b(x))$$
Le type de x est t_1 (type des arguments de f_a, mais il doit pouvoir être passé comme argument de f_b (de type t_3, donc
$$(type\_arg(f_a) = type(x) = t_1) \leq type\_arg(f_b) \Rightarrow t_1 \leq t_3$$

La relation $t_3 \leq t\_1$ est impossible car alors f_b ne pourrait recevoir un argument de type t_1 et l'appel $h(o_b, r)$ avec r de type t_1 serait alors incorrect.
exemple

En reprenant l'exemple sur les `point` et `point_colore` précédent, on obtient :

$$eq_{point} : point \rightarrow bool eq_{point\_colore} : point\_colore \rightarrow bool$$

et on s'aperçoit alors que pour que

$$point\_colore \leq point$$

il faudrait que

$$(point\_colore \rightarrow bool) \leq (point \rightarrow bool)$$

c'est à dire, avec la relation de contra-variance des types fonctionnels

$$point \leq point\_colore$$

ce qui est faux. Donc *point_colore* n'est pas un sous-type de *point*!!!

5.1.4 sous-typage et classes paramétrées

Les contraintes de type sur les paramètres de types d'une classe paramétrées. Cela permet de restreindre le polymorphisme paramétrique par le polymorphisme d'inclusion. Dans le but de construire des listes de `graphical_object` il est possible de préciser la contrainte de types `printable` par `graphical_object`. Ainsi pour toutes les classes pouvant être sous-typées en `graphical_object`, leurs instances respectives pourront être éléments de telles listes.

5.2 Style fonctionnel

Le style de la programmation objet est le plus souvent impératif. Un message est envoyé à un objet qui modifie physiquement son état interne (ses champs de données). Néanmoins il est aussi possible d'aborder la programmation objet par le style fonctionnel. L'envoi d'un message à un objet retourne un nouvel objet.

5.2.1 copie d'objets

On utilise pour cela l annotation `{< ... >}` qui retourne une copie de l'objet (self) dans lequel les valeurs certains champs de données sont changées.

```
class point (x_init,y_init) =
object
  val x = x_init
  val y = y_init
  method moveto (a,b) = {<x=a; y=b>}
  method rmoveto (dx,dy) = {<x=x+dx;y=y+dy>}
  method affiche () =
    begin
      print_string "( ";
      print_int x;
      print_string " , ";
      print_int y;
      print_string ")";
     end
end;;
```

qui possède l'interface suivante :

```
class point :
  int * int ->
  object ('a)
    val x : int
    val y : int
    method affiche : unit -> unit
    method moveto : int * int -> 'a
    method rmoveto : int * int -> 'a
  end
```

Les méthodes `moveto` et `rmoveto` retourne un objet. On peut donc envoyer à leur résultat un message, comme dans l'exemple suivant :

```
# let p = new point (2,3);;
val p : point = <obj>
# (p#rmoveto(10,10))#affiche();;
( 12 , 13)- : unit = ()
# p#affiche();;
( 2 , 3)- : unit = ()
```

La fonction`Oo.copy` retourne une copie d'un objet. Son type est le suivant :

```
(< .. > as 'a) -> 'a
```

5.2.2 avec les classes paramétrées

Le style fonctionnel peut utiliser les classes paramétrées avec la difficulté supplémentaire, due au sous-typage, de ne pas faire apparaître le type de `self` comme type résultat d'une méthode. C'est à dire, le type de `self` ne doit pas sortir de la classe.

On redéfinit la classe abstraite `['a] liste` de la manière suivante :

```
class virtual ['a] liste () =
object
  method virtual empty : unit -> bool
  method virtual cons  : 'a -> 'a liste
  method virtual head :  'a
  method virtual tail : 'a liste
  method virtual display : unit -> unit
end;;
```

où la méthode `cons` retourne une nouvelle liste.

La classe `cons` doit alors coercer le type de `self` dans la méthode `cons`.

```
class  ['a] cons (v ,l)  =
object (self)
  inherit ['a] liste ()
  constraint 'a = #printable
  val car = v
  val cdr = l
  method empty () = false
  method cons x = new cons (x, (self  : 'a #liste :> 'a liste))
  method head = car
  method tail = cdr
  method display () =
      begin
        car#print();
        print_string "  ::";
        self#tail#display()
    end
end;;
```

La classe `nil` utilisant le constructeur `cons` ne subit pas de modification sur les contraintes de types.

```
exception ListeVide;;
class  ['a] nil () =
object (self)
  inherit ['a] liste ()
  val nil = ()
  method empty () = true
  method cons x = new cons (x, new nil())
  method head = raise ListeVide
  method tail = raise ListeVide
  method display () = print_string "[]"
end;;
```

5.3 Interface

Les interfaces de classes sont en général inférées par le *typechecker* d'O'Caml, mais peuvent aussi être définies par une déclaration de types.

L'interface `interf_point` est déclarée de la manière suivante :

```
class type interf_point  =
object
```

```
  method get_x : int
  method get_y : int
  method moveto : (int * int ) -> unit
  method rmoveto : (int * int )  -> unit
  method print : unit -> unit
  method distance : unit -> float
end;;
```

```
class type interf_point =
  object
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
  end
```

Celle-ci peut dont être utilisée pour coercer le type d'une définition de classe.

```
# let f (x:interf_point) = x;;
val f : interf_point -> interf_point = <fun>
```

Ces interfaces ne peuvent masquer que les champs de variables d'instance et les méthodes privées. Elles ne peuvent en aucun cas masquer des méthodes abstraites ou des méthodes publiques. C'est là que leur limitaion apparaît. Néanmoins, elles utilisent aussi la relation d'héritage (entre interfaces).

En fait l'intérêt principal de pouvoir construire des interfaces de classes sans avoir à définir la classe, provient de l'utilisation des modules. Ainsi il sera possible de construire la signature d'un module, utilisant des types objets, uniquement en donnant la description des interfaces de ces classes.

5.4 Modules et objets

La programmation par modules et la programmation par objets ont des buts similaires : organisation logique d'un programme et compilation séparée. Elles permettent de faire de l'encapsulation de données, pour la réutilisation et la modification de composants, et d'étendre les fonctionnalités des composants. Il est à noter qu'avec les modules paramétrés il est possible de simuler les propriétés de l'héritage.

En effet soit le module `Point` de signature suivante :

```
module type POINT =
  sig
    type point
    val new_point : (int * int) -> point
    val get_x : point -> int
    val get_y : point -> int
    val moveto : point -> (int * int) -> unit
    val rmoveto : point -> (int * int) -> unit
    val affiche : point -> unit
    val distance : point -> float
  end;;
```

On peut construire un module `Point_colore` de la manière suivante :

```
module Point_colore = functor (P : POINT) ->
struct
  type point_colore = {p:P.point;c:string}
  let new_point_colore p c = {p=P.new_point p;c=c}
  let get_c self = self.c
  let get_x self = let super = self.p in P.get_x super
```

```
  let get_y self = let super = self.p in P.get_y super
  let moveto self = let super = self.p in P.moveto super
  let rmoveto self = let super = self.p in P.rmoveto super
  let distance self = let super = self.p in P.distance super
  let affiche self =
    let super = self.p in
    begin
      P.affiche super;
      print_string ("de couleur "^ self.c)
    end
end;;
```

La lourdeur des déclarations "héritées" peut être alégée par une procédure automatique de déclaration. Les déclarations récursives de méthodes pourraient s'écrirent par un seul `let rec ... and`. L'héritage multiple entraîne des foncteurs à plusieurs paramètres. Le coût de la redéfinition n'est pas plus grand que la liaison tardive.

La principale différence entre programmation modulaire et programmation objet en O'Caml provient du système de types. En effet la programmation par modules reste dans le système de types à la ML, i.e. du polymorphisme paramétrique (même code exécuté pour différents types de paramètres), alors que la programmation par objets et la liaison tardive entraîne un polymorphisme *ad hoc* (où l'envoi d'un message à un objet déclenche l'application de codes différents). Cela est particulièrement clair avec le sous-typage. Cette extension du système de types à la ML ne peut se simuler en ML pur. Il sera toujours impossible de construire des listes hétérogènes sans casser le système de types.

En conclusion la programmation modulaire et la programmation par objets sont deux réponses sûres (grace au typage) à l'organisation logique d'un programme, permettent la réutilisabilité et la modifiabilité de composants logiciels. La programmation objet en O'Caml permet le polymorphisme paramétrique (classes paramétrées) et d'*inclusion* (envoi de messages) grace à la liaison tardive et au sous-typage, avec des restrictions dues à l'égalité, ce qui facilite une programmation incrémentale. La programmation modulaire reste dans le domaine du polymorphisme paramétrique sans restriction avec liaison immédiate ce qui peut être utile pour l'efficacité de l'exécution.

5.5 Autres lectures

Ce cours s'inspire des mêmes documents que le cours précédent :

- Le cours de Francois Barthélémy et maria-Virginia Aponte au CNAM.
- La documentation d'O'Caml sur les objets;
- Une courte présentation des objets par Didier Remy
- Le cours de Didier Remy de Magistere MMFAI
  et de Roberto Di Cosmo au Magistere MMFAI .

Notes du cours 6

6 Systèmes à mémoire partagée

Objectifs :*Présentation du modèle à mémoire partagée de la programmation parallèle : section critique, exclusion mutuelle, sémaphores, processus légers en O'Caml*

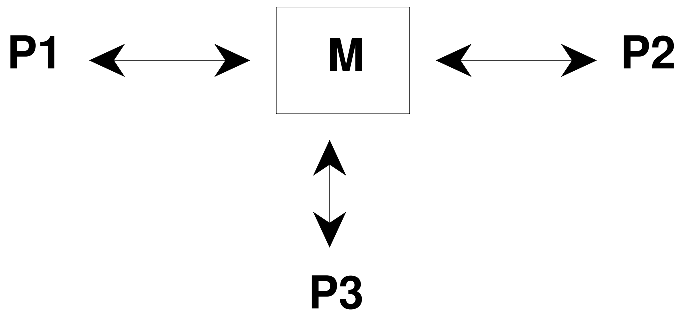
6.1 Généralités

On s'intéresse à deux grands modèles de programmation parallèle (ou simultanée) qui sont : les systèmes à mémoire partagée et les systèmes répartis (à mémoire répartie) que l'on étudiera au prochain cours. Chaque modèle met en valeur plusieurs notions fondamentales et étudie leurs relations. Les notions les plus importantes sont :

- la séquentialité (dépendance causale) : une instruction s'exécute après une autre;
- la concurrence (indépendance causale) : plusieurs instructions s'exécutent en "même temps";
- le non-déterminisme (une cause peut avoir plusieurs effets, mutuellement exclusifs) : un même programme ne termine pas ou termine en produisant des résultats différents;
- la synchronisation (plusieurs causes indépendantes doivent s'être produites avant que l'effet puisse avoir lieu) : attente d'une condition sur plusieurs processus;
- la communication (transfert d'informations) : envoi et réception d'informations d'un ou plusieurs processus à un ou plusieurs processus.

6.2 Systèmes à mémoire partagée

On considère un ensemble S de processus séquentiels P_i interagissant sur une mémoire commune (ou partagée) que l'on note $S = [P_1 \,||...||\, P_n]$. Ces processus peuvent être aussi bien physiquement indépendants (un processus correspond à un processeur) que simulés logiquement par un unique processeur (comme les Threads en O'Caml).



La communication dans ce modèle est implicite. L'information est transmise lors de l'écriture dans une zone de la mémoire partagée, puis quand un autre processus vient lire cette zone. Ce mécanisme est asynchrone, i.e. il ne nécessite pas que le récepteur soit prêt à écouter l'émetteur. Par contre la synchronisation doit être explicite, en utilisant des instructions élémentaires.

Sans synchronisation explicite, le résultat d'un programme est imprévisible. Par exemple, soit l'ensemble S de processus (avec x valant 0) défini ainsi : $S = [x := x + 1; x := x + 1 || x := 2 * x]$. Après l'exécution de S, x peut valoir 2,3, ou 4.

La synchronisation la plus simple est l'attente d'une condition. On la note *wait b*, où b est une expression booléenne. Un processus ne peut exécuter cette instruction que si b est vraie. En reprenant l'exemple précédent :
$S = [x := x + 1; x := x + 1 || wait(x = 1); x := 2 * x]$ on obtient comme valeur pour x que 3 ou 4. Par contre il est possible que le second processus reste bloqué s'il n'a testé x que pour les valeurs 0 ou 2.

Cela amène le problème de l'atomicité. Il peut être utile de manipuler l'atomicité de manière explicite. L'instruction *await b do P* attend que la condition b soit vraie pour exécuter les instructions de P de manière atomique dans le même état mémoire que le test de b.

6.3 Section critique, exclusion mutuelle

On appelle *section critique* une ressource qui ne doit être utilisée que par un processus au plus. Par exemple, on désire qu'un seul processus puisse utiliser une imprimante. C'est le cas du système Unix qui gère une queue d'impression sur les périphériques d'impression.

Pour cela les processus doivent s'exclure mutuellement de la section critique. On dit que l'activité A_1 du processus P_1 et l'activité A_2 du processus P_2 sont en *exclusion mutuelle* lorsque l'exécution de A_1 ne doit pas se produite en même temps que celle de A_2.

Le problème de l'exclusion mutuelle, dans un cas simple de deux processus, n'est pas si évident que cela. L'algorithme de Dekker est une solution qui fonctionne. Il utilise une variable globale `turn` que chaque processus peut consulter et changer dans la section critique.

```
let turn = ref 1;;
let c = Array.create 2 1;;

let crit i = ();;                   (* action dans la section critique *)
let suite i = ();;                  (*       hors section critique    *)

let p i =
  while true do
  begin
    c.(i)<-0;                        (* desire entrer dans la section critique *)
    while c.((i+1) mod 2) = 0 do     (* tant que l'autre processus
                                        desire aussi entrer dans la section critique *)
      if !turn = ((i+1) mod 2) then (* si c'est au tour de l'autre *)
      begin
        c.(i)<-1;                              (* abandon *)
        while !turn = ((i+1) mod 2) do done;   (* et attente de son tour *)
        c.(i)<-0                               (* puis reprise *)
      end;
    done;
    crit i;
    turn := ((i+1) mod 2);           (* passe le droit \'a l'autre processus *)
    c.(i)<-1;                        (* remise \'a 1 : sortie de la section critique *)
    suite i
  end;;

(* initialisation *)
c.(0)<-1;;
c.(1)<-1;;
```

```
turn:=1;;

(* lancement des processus *)

Thread.create p 0;;
Thread.create p 1;;
```

Les processus indiquent leur volonté d'entrer dans la section critique en mettant à 0 l'élément de tableau c les concernant. Après avoir marqué son élément de tableau le processus va regarder si l'autre processus est dans le même état (volonté d'entrer dans la section critique). Si ce n'est pas le cas, il entre dans la section critique, sinon il doit consulter l'arbitre ($turn$) qui indique à qui est le tour. Cet arbitre ne peut être modifié que dans la section critique (ici à la sortie). De ce fait, seul le processus étant entré dans la section critique modifiera l'arbitre à la fin de son travail en lui indiquant l'autre processus.

6.4 Sémaphores

Un sémaphore est une variable entière s ne pouvant prendre que des valeurs positives (ou nulles). Une fois s initialisé, les seules opérations admises sont : $wait(s)$ et $signal(s)$, notées respectivement $P(s)$ et $V(s)$. Elles sont définies ainsi :

- $wait(s)$: si $s > 0$ alors $s := s - 1$ ($await\ s\ do\ s := s - 1$), sinon l'exécution du processus ayant appelé $wait(s)$ est suspendue.
- $signal(s)$: si un processus a été suspendu lors d'une exécution antérieure d'un $wait(s)$ alors le réveiller, sinon $s := s + 1$.

s correspond au nombre de ressources d'un type donné.

remarques Un sémaphore ne prenant que les valeurs 0 ou 1 est appelé *sémaphore binaire*.

Les primitives $wait(s)$ et $signal(s)$ s'excluent mutuellement si elles portent sur le même sémaphore (l'ordre n'est donc pas connu).

La définition de *signal* ne précise pas quel processus est réveillé s'il y en a plusieurs.

On peut utiliser les sémaphores pour l'exclusion mutuelle. Les deux processus `p 1` et `p 2` sont exécutés en parallèle grâce à la bibliothèque de `threads` d'OCaml.

```
let s = ref 1;;

let p i =
  while true do
  begin
    wait(s);
    crit();
    signal(s);
    suite()
  end
;;

Thread.create p 1;;
Thread.create p 2;;
```

Dans cet exemple, si un processus veut entrer en section critique, il entrera en section critique si :
- il n'y a que 2 processus (si P_1 est suspendu alors P_2 est en section critique;
- si aucun processus ne s'arrête en section critique (si P_2 est dans `crit` alors il exécutera $signal(s)$.

Cette vérification ne fonctionne plus à partir de 3 processus. Il peut y avoir privation si le choix du processus se fait toujours en faveur de certains processus. Par exemple, si le choix s'effectue toujours en faveur du processus d'indice le plus bas, P_1 et P_2 pourraient se liguer pour se réveiller mutuellement, P_3 étant alors indéfiniment suspendu.

6.5 Le classique "dîner des philosophes"

Le "dîner des philosophes", du à Dijkstra, illustre les différents pièges du modèle à mémoire partagée.

L'histoire se passe dans un monastère reculé où 5 moines se consacrent exclusivement à la philosophie. Ils passeraient bien tout leur temps à la réflexion s'ils ne devaient manger de temps en temps. La vie d'un philosophe se résume en une boucle infinie : penser - manger. Ils possèdent une table commune ronde. Au centre se trouve un plat de spaguettis qui est toujours rempli. Il y s 5 assiettes et cinq fourchettes. Le philosophe qui veut manger sort de sa cellule, s'assoit à table, mange et retourne ensuite à ses pensées. Les spaguettis sont si enchevétrés qu'il faut deux fourchettes pour pouvoir les manger. Un philosophe ne peut utiliser que les deux fourchettes autour de son assiette.

Les problèmes posés sont :

- l'interblocage : chaque philosophe tient une fourchette et attend qu'une autre se libère;
- privation (ou famine) : un philosophe n'arrive jamais à obtenir 2 fourchettes.

6.6 Threads en O'Caml

La bibliothèque de threads (processus légers) d'O'Caml permet la programmation concurrente de processus pour un système à mémoire partagée. La communication s'effectue par modification des structures de données communes ou à travers des canaux de communications. On s'intéresse à la première approche. Cette bibliothèque ne peut s'utiliser qu'avec le compilateur en ligne dans sa version byte-code. Comme cette bibliothèque simule logiquement les processus par un seul programme, il ne faut pas attendre des gains de performance, mais seulement un gain d'expressivité du langage. Cette présentation est simplifiée dans la mesure où elle ne s'intéresse pas aux synchronisation d'écriture et de lecture, nis aux communications via les canaux.

6.6.1 les primitives importantes

module Thread

- `create  f a` crée le processus de l'application de `f` sur `a`;
- `self ()` retourne le processus courant et `id t` son identificateur.
- `exit ()` termine le processus courant et `kill t` le processus indiqué.

module Mutex

- `create ()` crée un verrou d'exclusion mutuelle (mutex);
- `lock m` capture un "mutex",`try_lock m` capture si possible un verrou, retourne `true` si cela est fait et `false` sinon, et `unlock` libère un verrou.

module Condition

- `create ()` crée une variable condition;
- `wait c m` libère `m` et suspend le processus appelant sur la variable condition `c`, `signal c` réveille un des processus suspendus sur la variable condition `c`, et `broadcast c` réveille tous les processus suspendus sur `c`.

6.6.2 Une solution pour le dîner

En utilisant la bibliothèque des processus légers de O'Caml, voici une solution nai"ve pour le dîner des philosophes :

```
let forks = Array.create 5 (Mutex.create());;
let i = ref 0 in
  Array.iter (fun x -> forks.(!i) <- Mutex.create(); incr i) forks;;

let think i = print_string (string_of_int(i)^" pense");print_newline();;
let eat i =  print_string (string_of_int(i)^" mange");print_newline();;

let philo i =
  print_string "debut du philosophe "; print_int i;print_newline();
  while true do
    think i;
    Mutex.lock(forks.(i));
    Mutex.lock(forks.((i+1) mod 5));
    eat i;
    Mutex.unlock(forks.(i));
    Mutex.unlock(forks.((i+1) mod 5))
  done
;;

List.iter (Thread.create philo) [0;1;2;3;4];;
while true do () done;;
```

Cette solution pose un problème d'interblocage, si tous les philosophes prennent leur fourchette droite "en même temps" ils restent en attente de la fourchette gauche.

Une solution serait de limiter la salle à manger à au plus quatre philosophes. Pour cela, il faut ajouter un sémaphore sur la salle à manger de capacité 4 au maximum.

On simule le sémaphore par les variables `room` et `mutex_room` qui correspondent respectivement à un compteur en zone critique.

```
let forks = Array.create 5 (Mutex.create());;
let i = ref 0 in
  Array.iter (fun x -> forks.(!i) <- Mutex.create();incr i) forks;;

let room = ref 0;;
let mutex_room = Mutex.create();;

let think i = print_string (string_of_int(i)^" pense");print_newline();;
let eat i =  print_string (string_of_int(i)^" mange");print_newline();;

let philo i =
  print_string "debut du philosophe "; print_int i;print_newline();
  while true do
    think i;
    Mutex.lock mutex_room;
    if !room < 4 then
    begin
      room := !room +1;
      Mutex.unlock mutex_room;
      print_string ("avec " ^string_of_int (i)^ " on est "^(string_of_int !room));
      print_newline();
      Mutex.lock(forks.(i));
      Mutex.lock(forks.((i+1) mod 5));
```

```
      eat i;
      Mutex.unlock(forks.(i));
      Mutex.unlock(forks.((i+1) mod 5));
      Mutex.lock mutex_room;
      room := !room - 1
    end;
    Mutex.unlock mutex_room
  done
;;

List.iter (Thread.create philo) [0;1;2;3;4];;
while true do () done;;
```

6.7 Relations entre threads

Les différentes `threads` d'un programme peuvent être en relation selon l'un des 4 modèles suivants :

- sans relations
- en relation mais sans synchronisation
- en relation avec exclusion mutuelle
- en relation avec exclusion mutuelle et communication (`wait/signal`

6.8 un producteur/consommateur

Cet exemple décrit un producteur/consommateur. Le programme principal comporte un "producteur" et 3 clients. Le producteur envoie des produits dans une boutique qui seront pris par les clients. Le stockage de la boutique étant limité, le producteur devra attendre qu'il y ait une place libre pour déposer son produit. Si ce n'est pas le cas, il se met en attente (`wait`) d'un signal indiquant une modification de l'état de la boutique. De même un client attendra qu'il y ait au moins un produit pour pouvoir le prendre. Dans tout les cas la boutique correspond à la zone d'exclusion mutuelle : on ne peut pas déposer ou prendre en même temps un produit.

```
class product (s:string) =
object
  val name = s
  method get_name = name
end;;

let m = Mutex.create();;
let c = Condition.create();;

class shop n =
object(self)
  val  mutable size =n;
  val  mutable buffer = ([||] : product array)
  val  mutable ip = 0
  val  mutable ic = 0

  initializer buffer<-Array.create 12 (new product "empty")

  method display1 () =
    print_string("enter product : [("^
```

```
 (string_of_int (ip mod size))^
 ")]"^
 ((buffer.(ip mod size))#get_name)^
 "");
    print_newline()

  method display2 () =
    print_string("exit product : [("^
 (string_of_int (ic mod size))^
 ")]"^
 ((buffer.(ic mod size))#get_name)^
 "");
    print_newline()

  method put p =
    Mutex.lock m;
    while (ip-ic+1 > Array.length(buffer)) do Condition.wait c m done;
    buffer.(ip mod size) <- p;
    self#display1();
    ip <- ip+1;
    Mutex.unlock m;
    Condition.signal c

  method get () =
    Mutex.lock m;
    while(ip == ic) do Condition.wait c m done;
    self#display2();
    let r = buffer.(ic mod size) in
      ic<- ic+1;
      Mutex.unlock m;
      Condition.signal c;
      r
end;;


class consumer sh na =
object(self)
  val a_shop = sh
  val name = na

  method run () =
   print_string"...";print_newline();
    while true do
      let p = a_shop#get() in
self#display(p);
Thread.delay(Random.float(3.0))
    done

  method display (p:product) =
    print_string("Get : "^
p#get_name^
" from "^
name ^
"\n");
    print_newline()
```

```
end;;

class producer sh  =
object(self)
  val a_shop = sh
  val mutable num = 0

  method run () =
    while true do
      let p = self#make_product() in
a_shop#put(p);
self#display(p);
Thread.delay(Random.float(1.0))
    done

  method display p =
    print_string("Put : "^
p#get_name^
"\n");
    print_newline()

  method make_product() =
    let p = new product("Product "^(string_of_int num)) in
      num <- num + 1;
      p

end;;


let ash = new shop 12;;
let p = new producer ash;;
let c1 =  new consumer ash "cl-1";;
let c2 =  new consumer ash "cl-2";;
let c3 =  new consumer ash "cl-3";;

let goc (c:consumer) = c#run();;
let gop (p:producer) = p#run();;

Thread.create gop p;;
Thread.create goc c1;;
Thread.create goc c2;;
Thread.create goc c3;;

Thread.sleep();;
```

6.9 Autres lectures

Ce cours s'inspire des documents suivants :

- "Processus Concurrents", Ben Ari, Masson 1986;
- "Concurrent Programming : Principles and practices", G. Andrews, Benjamin Cumming, 1991;
- Sémantique du parallélisme : un tour d'horizon en PostScript compressé de Luc Bougé.
- Le cours de Jean-Jacques Levy à l'X sur concurrence et mémoire partagée
- La page de Concurrent ML extension de SML/NJ.

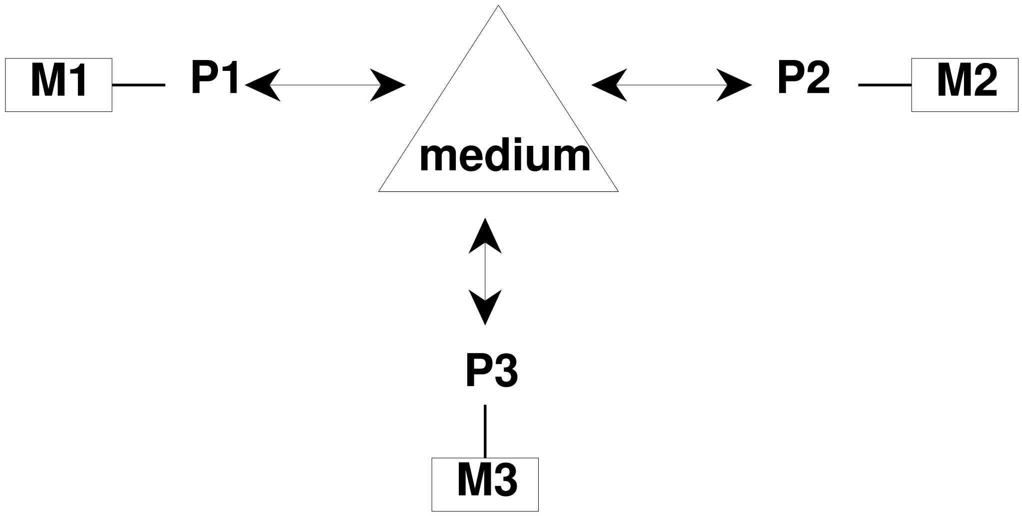
Notes du cours 7

7 Systèmes à mémoire distribuée

Objectifs :*Présentation du modèle à mémoire distribuée de la programmation parallèle : communications, interneteries, modèle client/serveur, bibliothèque Unix en O'Caml*

7.1 Systèmes à mémoire distribuée ou systèmes répartis

Dans ce modèle chaque processus séquentiel P_i possède sa propre mémoire privée M_i. Il est le seul à y avoir accès. Les processus doivent alors communiquer pour transférer de l'information. On suppose alors qu'il y a un medium assurant ces transferts. La difficulté de ce modèle provient de l'implantation du medium. Les programmes s'en chargeant s'appellent des *protocoles*.



Ceux-ci sont organisés en couche. Les protocoles de haut niveau, implantant des services élaborés, utilisent les couches de plus bas niveaux (voir les 7 couches du modèle ISO).

Ce modèle est valable dans le cas de parallélisme physique (réseau d'ordinateurs) ou logique (processus Unix communiquant par "pipes" ou threads O'Caml communiquant par canaux). Il n'y a pas de valeurs globales connues par tous les processus (comme un temps global). La seule contrainte sur le temps est l'impossibilité de recevoir un message avant son émission.

Dans ce modèle la communication est explicite alors que la synchronisation est implicite (elle est en fait produite par la communication). Ce modèle est le dual du précédent.

7.1.1 modèles de communications

- un-à-un (point-à-point) : communication d'un processus à un autre; les autres processus ignorent cette communication. Les deux primitives sont "envoi d'une valeur sur un canal" et "réception d'une valeur d'un canal".
- un-à-tous (diffusion) : communication d'un processus à tous les processus. Les primitives de communication sont : "envoi d'une valeur à tous" et "réception d'une valeur".

- tous-à-tous (diffusion) : communication de tous les processus à tous les processus. La réception tient compte alors des différentes valeurs envoyées.

7.1.2 types de communications

- synchrone : le transfert d'informations n'est possible que lors d'une synchronisation globale des processeurs émetteur et récepteur. L'émission et la réception peuvent être bloquantes.
- asynchrone : le medium peut stocker des messages en vue de leur acheminement futur. Il faut donc spécifier la capacité de stockage, l'ordre d'acheminement, les délais de transmissions et la fiabilité de transmission. L'émission est non bloquante.
- évanescent : l'émission est non bloquante et le medium ne peut pas stocker de messages. Le message émis est reçu par les processus prêt à le recevoir et perdu pour les autres.

7.2 Interneteries

Internet est un réseau de réseau (interconnexion de réseaux). Certaines machines ont néanmoins un rôle particulier : les passerelles (pour le passage d'un réseau à un autre) et les routeurs (indiquant la route à suivre). Conceptuellement l'interconnexion se fait au niveau des réseaux et non pas des machines. Deux machines quelconques connectées sur Internet peuvent communiquer. Le réseau devient une seule entité. Différents types de machines, systèmes cohabitent sur Internet. Elles parlent toutes les protocoles IP (et UDP/TCP).

7.2.1 protocoles Internet

L'organisation du réseau Internet est la suivante :

```
----------------
| APPLICATIONS |
----------------
   ^    |
   |    |
   |    v
----------------
|   UDP / TCP  |
----------------
   ^    |
   |    |
   |    v
----------------
| IP/Interfaces|
----------------
   ^    |
   |    |
   |    v
----------------
|    MEDIAS    |
----------------
```

Le protocole IP est le protocole de bas niveau. L'unité de transfert est le datagramme IP. C'est un protocole non fiable : il n'assure ni le bon ordre, ni le bon port, ni la non duplication des datagrammes transmis. Il traite juste le routage d'un datagramme et la gestion des erreurs quand un datagramme n'a pas être transmis. Un datagramme contient un entête et des données. Dans l'entête apparaît les adresses du destinataire et de l'expéditeur du datagramme.Chaque machine sur Internet possède une adresse unique. Ces adresses sont codées sur 32 bits (IPv4). Par exemple l'adresse `132.227.60.30` contient 4 champs comportant des valeurs de 0 à 255.

Au dessus d'IP, deux protocoles permettent des transmissions de plus haut niveau : UDP (User Datagram Protocol) et TCP (Transfert Control Protocol). UDP est un protocole sans connexion et non fiable (il sert à multiplexer les transmissions). TCP est un protocole orienté connexion et fiable. Pour cela il doit gérer les acquittements de paquets et optimiser la transmission (technique de fenêtrage).

Les services standards (applications) d'Internet utilisent le plus souvent le modèle client/serveur. Le serveur est un programme offrant un service spécifique. Il gère les requêtes des *clients* en établissant une connexion ou non selon le protocole. Il y a une asymétrie entre le client et le serveur. Ces services implantent des protocoles de plus haut niveau. Parmi les services standards, on peut nommer :

- FTP (File Transfert Protocol)
- Telnet (Terminal Transfert Protocol)
- SMTP (Simple Mail Transfert Protocol)
- HTTP (Hyper Text Transfert Protocol)

D'autres services utilisent ce modèle client/serveur :

- NFS (Network File System)
- X-Window
- les services Unix : rlogin, rwho ...

La communication entre applications s'effectuent via des prises (sockets) de communication. Elles permettent la communication entre des processus ne résidant pas forcément sur une même machine. Différents processus peuvent lire et écrire dans cette voie de communication.

7.3 Communications en O'Caml

Il existe trois possibilités d'utiliser la communication entre processus en O'Caml. La première utilise les communications entre threads. La deuxième utilise les primitives `fork` et `pipe` d'Unix. Ces deux premières méthodes utilisent un modèle logique de concurrence. Il n'y aura pas d'améliorations de performances, tous les processus tournent sur le même processeur. La troisième possibilité utilise les prises de communication (sockets) d'Unix. La communication peut alors s'effectuer entre différentes machines physiques.

7.3.1 canaux des threads

Le module `Event` de la bibliothèque des `Threads` d'O'Caml autorise la communication entre threads à travers des canaux de communication. Deux types sont définis :

```
type 'a channel
type 'a event
```

correspondant au type des canaux et des valeurs transmises.

Les communications s'effectuent principalement par les fonctions suivantes :

```
new_channel : unit -> 'a channel
send : 'a channel -> 'a -> unit evant
receive : 'a channel -> 'a event
sync : 'a event -> 'a
choose : 'a event list -> 'a event
select : 'a event list -> a
```

7.3.2 bibliothèque Unix : fork et pipe

La bibliothèque Unix d'O'Caml implante les principaux appels système de la bibliothèque Unix. Une des facilités de cette bibliothèque est d'utiliser la boucle de toplevel pour implanter ses fonctions système. Le typage facilite aussi la programmation système. Il n'est plus aussi nécessaire de pratique la lecture du man, le type de la fonction est bien souvent suffisamment parlant.

Parmi les nombreuses fonctions de cette bibliothèques, on retrouve les appels système `fork` pour la duplication d'un processus, les tuyaux (`pipe`) de communications et le duplicateur (`dup2`) de description de fichiers.

```
match fork () with
  0 -> (* code du fils *)
| _ -> (* code du pere *)
```

La communication entre processus

```
pipe : unit -> Unix.file_descr * Unix.file_descr
dup2 : Unix.file_descr -> Unix.file_descr -> unit
```

7.3.3 bibliothèque Unix : sockets

Les principaux types sont les suivants :

```
type socket_domain = PF_UNIX | PF_INET;;

type socket_type = SOCK_STREAM | SOCK_DGRAM |
                   SOCK_SEQPACKET | SOCK_RAW;;

type sockaddr =
  ADDR_UNIX of string | ADDR_INET of inet_addr * int
```

Les principales fonctions d'établissement d'une prise et de communications sont les suivantes :

```
socket : Unix.socket_domain -> Unix.socket_type -> int -> Unix.file_descr
connect : Unix.file_descr -> Unix.sockaddr -> unit
close : Unix.file_descr -> unit
bind : Unix.file_descr -> Unix.sockaddr -> unit
listen : Unix.file_descr -> int -> unit
accept : Unix.file_descr -> Unix.file_descr * Unix.sockaddr
read : Unix.file_descr -> string -> int -> int -> int
write : Unix.file_descr -> string -> int -> int -> int
```

où `socket` crée un `file_descr` Unix. Pour créer une prise on utilisera l'appel suivant :

```
let prise = socket PF_INET SOCK_STREAM 0;;
```

où `PF_INET` correspond au domain Internet, `SOCK_STREAM` pour un flot d'octets fiable et l'entier `0` au protocole IP.

La connexion à un serveur s'effectue par la fonction `connect`. L'établissement d'un service passe par les étapes suivantes :

- association d'une adresse à une prise (`bind`) pouvant être utilisée par la suite de l'extérieur;
- acceptation des connexions externes (`listen`);
- réception de demande de connexion (`accept`).

Dès qu'une connexion est établie, les fonctions d'E/S (`read` et `write`) peuvent être utilisées sur la prise.
Les fonctions suivantes

```
gethostbyname : string -> Unix.host_entry
getservbyname : string -> string -> Unix.service_entry
```

permettent à partir d'un nom de machine d'obtenir d'une part la description complète d'une machine (incluant son adresse) etr d'autre part la description d'un service (incluant son numéro de port).

7.4 Exemples de client/serveur

7.4.1 serveur universel

Le schéma classique d'un serveur est le suivant :

```
creation de ls socket (socket)

attachement de la socket (bind)

ouverture du service (listen)

attente de connexion (accept)

  creation d'un processus (fork)


fils : boucle sur l'attente de la connexion
pere : traite la demande
```

code du serveur

```
let main () =
  let port =  int_of_string argv.(1)
  and args = Array.sub argv (Array.length argv - 2) in
  let sock =
    socket PF_INET SOCK_STREAM 0 in
  let mon_adresse =
   (gethostbyname(gethostname())).h_addr_list.(0) in
  bind sock (ADDR_INET(mon_adresse, port));
  listen sock 3;
  while true do
    let (s,app_adr) = accept sock in
    begin
      match app_adr with
        ADDR_INET(host,_) ->
          print_string ("Connexion depuis "^ string_of_inet_addr host);
          print_newline()
      | ADDR_UNIX _ -> ()
    end;
    match fork() with
     0 -> if fork() <> 0 then exit 0;
          List.iter (dup2 s) [stdin;stdout;stderr];
          close s;
          () (* travail \'a effectuer *)
    | _ -> wait ();
           close s
  done
;;

handle_unix_error main ();;
```

La technique du double "fork" permet d'éviter de laisser des processus "zombies" (en attente de transmission de leur code retour à leur père). Le fils termine par `exit` juste après le deuxième `fork`. Le petit fils devient donc orphelin, et est adopté par le processus `init` qui possède une boucle infinie de `wait`, et fera disparaître le processus petit fils dès qu'il termine.

code du client

```
let main () =
  let serveur = argv.(1)
  and port = int_of_string (argv.(2)) in
  let args = Array.sub argv (Array.length argv - 3) in
  let serveur_adr =
    try
      inet_addr_of_string serveur
    with Failure _ ->
      try
        (gethostbyname serveur).h_addr_list.(0)
      with Not_found ->
        prerr_endline (serveur ^ " : serveur inconnu");
        exit 2
   in
   let sock =
     socket PF_INET SOCK_STREAM 0
   in
     connect sock (ADDR_INET(serveur_adr,port));
     match fork() with
       0 -> (* travail du fils a effectuer *)
            shutdown sock SHUTDOWN_SEND; (* fermeture en ecriture *)
            exit 0
     | _ -> (* travail du pere *)
            close sock;
            wait()
;;
```

Ce schéma père/fils permet de répartir les rôles en envoi/réception sur la prise. Ici le fils écrit sur la prise la requête et le père reçoit la réponse. Cette architecture prend du sens si le fils doit envoyer plusieurs requêtes, le père recevra les réponses des premières requêtes au fur et à mesure de leur traitement.

7.5 Threads et bibliothèque Unix

L'utilisation conjointe de la bibliothèque de processus légers et de la bibliothèque `Unix` provoque le blocage de toutes les "threads" actives si un appel système ne répond pas immédiatement, en particulier les lectures sur un descripteur de fichiers, incluant donc ceux créés par `socket`.

Pour éviter ce désagrément, le module `ThreadUnix` réimplante la plupart des fonctions de la bibliothèque Unix. Les fonctions définies dans ce module ne bloqueront que la thread qui effectue cet appel.

7.6 Autres lectures

Ce cours s'inspire des documents suivants :

- "Processus Concurrents", Ben Ari, Masson 1986.
- Sémantique du parallélisme : un tour d'horizon en PostScript compressé de Luc Bougé.
- "Communication sous Unix", Jean-Marie Rifflet, EdiSciences.
- "Linux 2.0:, Rémy Card, Eric Dumas, Franck Mevel, Eyrolles, 1996.
- "Programmation du Système Unix en Caml Light", Xavier Leroy, RT 147, INRIA, 1992.
- "TCP/IP", Douglas Cormer, InterEditions.
- "Internet sous Unix BSD", Yves Legrand-Gérard, INET'96.

La page du système Ensemble montre d'une boite à outils de communications développée en O'Caml.

Liste des liens

Voici la liste des liens rencontrés dans ces notes de cours :

```
{La documentation d'O'Caml sur les modules}\\
  {http://cadillac.lip6.fr/~emmanuel/Private/documents/ocaml-1.05/node4.html}
{La documentation d'O'Caml sur les objets}\\
  {http://cadillac.lip6.fr/~emmanuel/Private/documents/ocaml-2.00/node3.html}
{Emmanuel Chailloux}{http://cadillac.lip6.fr/~emmanuel}
{Pascal Manoury}{http://cadillac.lip6.fr/~manoury}
{Bruno Pagano}{http://cadillac.lip6.fr/~pagano}
{TheCours}{http://nephi.unice.fr/CoursHTML/cours.html}
{en PostScript gzipp\'e}{http://pauillac.inria.fr/~xleroy/public/caml-special-light-rr.ps.gz}
{Concurrent ML}{http://cm.bell-labs.com/cm/cs/who/jhr/sml/cml/index.html}
{Caml}{http://pauillac.inria.fr/caml/index-fra.html}
{Le cours de Didier Remy de Magistere MMFAI}{http://pauillac.inria.fr/~remy/classes/magistere/}
{Une courte pr\'esentation des objets}{http://pauillac.inria.fr/~remy/objectdemo.html}
{en PostScript}{http://pauillac.inria.fr/~xleroy/dea/cours1.ps}
{"Programmation du Syst\`eme Unix en Caml Light"}{http://pauillac.inria.fr/~xleroy/publi/unix-in-caml.ps.gz}
{Ensemble}{http://simon.cs.cornell.edu/Info/Projects/Ensemble/index.html}
{CEDRIC}{http://tulipe.cnam.fr/cours/cours1.html}
{Le cours}{http://tulipe.cnam.fr/personne/aponte/ocaml.html}
{WWW}{http://web.urec.fr/docs/WWW}
{http://www-spi.lip6.fr/~emmanuel/Public/enseignement/pod\_98\_index.html}\\
  {http://www-spi.lip6.fr/~emmanuel/Public/enseignement/pod\_98\_index.html}
{Scol}{http://www.cryo-networks.com/}
{de Roberto Di Cosmo au Magistere MMFAI}{http://www.dmi.ens.fr/users/dicosmo/CourseNotes/OO/}
{en PostScript compress\'e}{http://www.ens-lyon.fr/~bouge/Publis/Horizon_88.ps.Z}
{en PostScript compress\'e}{http://www.ens-lyon.fr/~bouge/Publis/Horizon_88.ps.Z} de Luc Boug\'e.
{Approche Fonctionnelle de la Programmation}{http://www.ens.fr/~cousinea/Book/livre.html}
{1984}{http://www.ens.fr/~cousinea/Caml/caml_history.html}
{premier langage}{http://www.ens.fr/~cousinea/LF_ENS/LF_ENS.html}
{SPP}{http://www.ens.fr/~dicosmo/DEA/SPP/},
{manuel illustr\'e}{http://www.grr.ulaval.ca/grrwww/manuelhtml.html}
{26}{http://www.imaginet.fr/ime/html32.htm}.
{24}{http://www.imaginet.fr/ime/htmlpub.htm}
{UNGI 97}{http://www.imaginet.fr/ime/nethtm2.htm}
{Documentations en ligne des langages O'Caml et Java}{http://www.infop6.cicrp.jussieu.fr/logiciels}.
{concurrence et m\'emoire partag\'ee}\\
  {http://www.polytechnique.fr/poly/~levy/poly/majifa-syst-96/cours/cours4.html}
{Consortium W3}{http://www.w3.org/MarkUp}
```

Table des matières