TEP (NI512) : COURS 3

Typage et Polymorphisme

Emmanuel Chailloux

PLAN

- Inférence de types d'un mini-ML
  - Noyau fonctionnel
  - traits impératifs
- Objets et types en OCaml
  - structuration en classes, sous typage structurel
  - relations d'héritage et de sous-typage

Inférence de types

Algorithme permettant de déterminer le type d'une expression

- utilisant un système de règles
- construisant l'arbre de preuve du type

Notation des règles:

1. $C \vdash e : \tau$
   l'expression e a le type τ dans le contexte de types C
2. $$\frac{C \vdash e_1 : \tau_1 \quad C \vdash e_2 : \tau_2}{C \vdash (e_1, e_2) : \tau_1 \ * \ \tau_2}$$

   Si $C \vdash e_1 : \tau_1$ et $C \vdash e_2 : \tau_2$ alors $C \vdash (e_1, e_2) : \tau_1 \ * \ \tau_2$

Syntaxe d'un mini-ML

```
constante ::= entier | flottant | booleen

expr ::= constante
       | []
       | ident
       | expr , expr
       | expr :: expr
       | if expr then expr else expr
       | expr expr
       | function ident -> expr
       | let ident = expr in expr
       | let rec ident = expr in expr
       | ( expr )
```

Types de mini-ML

L'ensemble des types ($\mathcal{T}$) de mini-ML est construit à partir de ($\mathcal{C}$) (ctes), de ($\mathcal{P}$) (variables) et de 3 constructeurs de types ($\rightarrow$, $*$ et *list*).

- types de mini-ML : $\mathcal{T}$
  - constante de type (int, float,...) : si $\alpha \in \mathcal{C}$ alors $\alpha \in \mathcal{T}$;
  - variable de type $\in \mathcal{P}$:si $\alpha \in \mathcal{P}$ alors $\alpha \in \mathcal{T}$;.
  - type produit : si $\tau_1, \tau_2 \in \mathcal{T}$ alors $\tau_1 \ * \ \tau_2 \in \mathcal{T}$;
  - type liste : si $\tau \in \mathcal{T}$ alors τ *list* $\in \mathcal{T}$;
  - type fonctionnel : si $\tau_1, \tau_2 \in \mathcal{T}$ alors $\tau_1 \ \rightarrow \ \tau_2 \in \mathcal{T}$;
- schémas de type : $\mathcal{S}$
  - si $\tau \in \mathcal{T}$ alors $\tau \in \mathcal{S}$
  - si $\sigma \in \mathcal{S}, \alpha \in \mathcal{P}$ alors $\forall \alpha.\sigma \in \mathcal{S}$

Synthèse par unification

Une substitution est une application $\mathcal{P} \to \mathcal{T}$.
On peut appliquer une substitution s à un terme $\sigma \in \mathcal{T}$, ce qui donne un terme σs :

- $xs = s(x)$
- $(\sigma \ \to \ \tau)s = (\sigma s) \ \to \ (\tau s)$
- $(\sigma \ * \ \tau)s = (\sigma s) \ * \ (\tau s)$
- $(\sigma \ \mathit{list})s = (\sigma s)\mathit{list}$

Si $\sigma, \tau \in \mathcal{T}$, unifier σ et τ, c'est trouver une substitution s telle que $\sigma s = \tau s$.

> *Pour tous $\sigma, \tau \in \mathcal{T}$ unifiables, il existe un unificateur qui est plus général que tous les autres. On l'appelle unificateur principal, et on le note $UP(\sigma, \tau)$.*

Règles de typage d'un mini-ML (1)

$$\text{(ConstInt)}\quad C \vdash \textit{nombre entier} : \textit{Int}$$

$$\text{(ConstFloat)}\quad C \vdash \textit{nombre flottant} : \textit{Float}$$

$$\text{(ConstString)}\quad C \vdash \textit{chaine} : \textit{String}$$

$$\text{(ConstBool)}\quad C \vdash \textit{booleen} : \textit{Bool}$$

$$\text{(ConstUnit)}\quad C \vdash () : \textit{Unit}$$

Règles de typage d'un mini-ML (2)

(Var)

$$x : \sigma, C \vdash x : \tau \quad \tau = \mathit{instance}(\sigma)$$

(Pair)

$$\frac{C \vdash e_1 : \tau_1 \quad C \vdash e_2 : \tau_2}{C \vdash (e_1, e_2) : \tau_1 \ * \ \tau_2}$$

(List)

$$\frac{C \vdash e_1 : \tau \quad C \vdash e_2 : \tau \ \mathit{list}}{C \vdash (e_1 \ :: \ e_2) : \tau \ \mathit{list}}$$

Règles de typage d'un mini-ML (3)

(If)

$$\frac{C \vdash e_1 : Bool \quad C \vdash e_2 : \tau \quad C \vdash e_3 : \tau}{C \vdash (if\ e_1\ then\ e_2\ else\ e_3) : \tau}$$

(App)

$$\frac{C \vdash e_1 : \tau' \to \tau \quad C \vdash e_2 : \tau'}{C \vdash (e_1\ e_2) : \tau}$$

(Abs)

$$\frac{x : \tau_1, C \vdash e : \tau_2}{C \vdash (function\ x\ ->\ e) : \tau_1 \to \tau_2}$$

Déclarations locales

Let et LetRec:

(Let)

$$\frac{C \vdash e_1 = \tau_1 \quad \sigma = \mathit{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

(LetRec)

$$\frac{(x : \alpha), C \vdash e_1 = \tau_1 \quad \alpha \notin V(C) \quad \sigma = \mathit{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2}{C \vdash \mathit{let\ rec}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

Seules les déclarations introduisent le polymorphisme.

Algorithme de synthèse (1)

- procédé inverse
- construction de l'arbre de preuve du type

exemple: une fonction `type` qui prend un contexte C et une expression e et retourne un type τ et une substitution s
type (C,`if e1 then e2 else e3`) $\rightarrow \tau, s$

- type(C,e_1) $\rightarrow \tau_1$,s_1
  type (s_1(C),e_2)$\rightarrow \tau_2$,s_2 type ($s_2(s_1$(C)),e_3)$\rightarrow \tau_3$,s_3
- u_1=UP(τ_1,Bool), u_2=UP($u_1(\tau_2)$,$u_1(\tau_3)$), τ=$u_2(u_1(\tau_2))$

2 types d'erreur:

1. clash sur les constantes, ou les constructeurs
2. α est substitué par un type contenant α

Algorithme de synthèse (2)

En Objective Caml:

- application immédiate de la substitution (modification physique)
- clash
  - entre constructeurs

```
1 # 1 :: [] = 1 , [] ;;
2 This expression has type int but is here used with ↩
      type int list
3 # ( 1 :: [] ) = ( 1 , [] ) ;;
4 This expression has type int * 'a list but is here ↩
      used with type int list
```

  - occur check

```
1 # let delta x = x x ;;
2 This expression has type 'a -> 'b but is here used ↩
      with type 'a
3 # let omega = (function x -> x x) (function y -> y y )↩
      ;;
4 This expression has type 'a -> 'b but is here used ↩
      with type 'a
```

Introduction des traits impératifs (1)

Danger des modifications physiques en présence du polymorphisme :

```
let x = ref (fun x -> x) in
     x:=(fun x -> x + 1);
          !x true;;
```

Introduction des traits impératifs (2)

$\Rightarrow$ 2 types d'expressions :

- expression non expansive : variables, constructeurs, abstraction
- expression expansive : application, ref

Seules ces dernières peuvent :

- engendrer une exception
- transgresser le système de types

$\Rightarrow$ utiliser des variables de type faibles.

Modification du système (1)

(Ref) pour les valeurs mutables

$$\frac{(C \vdash e : \tau}{C \vdash \mathit{ref}\ e : \tau\ \mathit{ref}}$$

(LetRec) e_1 de la forme **function**

$$\frac{(x : \alpha), C \vdash e_1 = \tau_1 \quad \alpha \notin V(C) \quad \sigma = \mathit{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \mathit{let\ rec}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

Modification du système (2)

(Let1) si $e_1 \in$ Valeur

$$\frac{C \vdash e_1 = \tau_1 \quad \sigma = \mathit{generalize}(\tau_1, C) \quad (x : \sigma), C \vdash e_2 : \tau_2}{C \vdash \mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

(Let2) si $e_1 \notin$ Valeur

$$\frac{C \vdash e_1 = \tau_1 \quad (x : \tau_1), C \vdash e_2 : \tau_2}{C \vdash \mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

Valeur correspond aux expressions non expansives : constantes, variables et abstraction

Modification du système (3)

Conséquence: l'application n'est plus généralisée.

Intérêt: même type en fonctionnel ou en impératif.

Inconvénient: certains programmes ancciennement typables ne le sont plus

Typage en Objective Caml

Extraction d'une partie des sources:

```
let rec is_nonexpansive exp =
  match exp.exp_desc with
Texp_ident(_,_) ->true |Texp_constant _ ->true
  | Texp_let(rec_flag, pat_exp_list, body) ->
      List.for_all (fun (pat, exp) ->
                    is_nonexpansive exp)
        pat_exp_list & is_nonexpansive body
  | Texp_function _ -> true
  | Texp_tuple el ->
      List.for_all is_nonexpansive el
  | Texp_construct(_, el) ->
      List.for_all is_nonexpansive el
  | Texp_record(lbl_exp_list, opt_init_exp) ->
      List.for_all
        (fun (lbl, exp) ->  lbl.lbl_mut =
          Immutable & is_nonexpansive exp)
        lbl_exp_list &&
      (match opt_init_exp with None -> true
         | Some e -> is_nonexpansive e)
  | Texp_field(exp, lbl)->is_nonexpansive exp
  | Texp_array [] -> true
  | _ -> false
```

Application non généralisée (1)

Rejet: de programmes fonctionnels corrects

```
1  # let id = function x -> x;;
2  val id : 'a -> 'a = <fun>
3
4  # let i = id id;;
5  val i : '_a -> '_a = <fun>
6
7  # i 3;;
8  - : int = 3
9
10 # i;;
11 - : int -> int = <fun>
12
13 # i true;;
14 This expression has type bool
15 but is here used with type int
```

Application non généralisée (2)

Solution: ajouter un paramètre

```
1  # let j y = id id y;;
2  val j : 'a -> 'a = <fun>
3
4  # j 3;;
5  - : int = 3
6
7  # j true;;
8  - : bool = true
```

Plan :

- Généralités
- Classe, objet, appel de méthode
- Héritage, liaison tardive
- Visibilité
- Classes et méthodes abstraites
- Objets et types, types ouverts
- Héritage multiple, classes paramétrées
- Exemples

Programmation par objets (1)

Qu'est-ce qu'un objet ?

regroupement de données et de traitements qui s'y appliquent

- entité autonome réagissant à des messages
- un objet est une entité possédant un état modifiable par envoi de messages
- un objet est une entité logicielle réutilisable composée d'un état (le plus souvent opaque) et ayant un comportement propre.

$\Rightarrow$ une valeur ayant un état propre et une interface de communication

Programmation par objets (2) : caractéristiques principales

- encapsulation
  - vision types abstraits de données
    - spécification/interface et réalisation/implantation
    - masquer des traitements, des données, des types, ...
  - vision client (utilisateur) / fournisseur (concepteur)
    - les qualifieurs et l'interface définissent le protocole de communication
- réutilisabilité
  - héritage
    - récupération et spécialisation de code existant
  - liaison tardive
    - détermination à l'exécution de la méthode à employer
  - principe de subsomption
    - utilisation d'un objet d'une certaine classe/spécification à la place et lieu d'un objet d'une autre classe/spécification

Programmation par objets (3) : caractéristiques principales

- généricité du code
  - polymorphisme objet (ou d'inclusion),
  - polymorphisme ad hoc (surcharge),
  - polymorphisme paramétrique (génériques)

- Facilite la modélisation des relations entre les entités d'un programme (classes, interfaces, objets, ...)

Les langages à objets (1) : historique

historique des langages : poster O'Reilly

`http://oreilly.com/news/graphics/prog_lang_poster.pdf`

- années 80 : la recherche
  - communauté scientifique : langages de programmation + IA
  - langages Simula, SmallTalk (80)
- années 90 : l'industrie
  - langages ou extensions objets utilisés dans l'industrie : SmallTalk, C++ (ATT), Objective C (NextStep, puis MacOSX), CLOS, Delphi (Borland), Java (Sun 95), C# (Microsoft), Python, Javascript (NetScape 95), Ruby, ...
  - émergence du génie logiciel : langage de modélisation (UML)
- années 2000 : méthodes et outils
  - Programmation générique typée, tests unitaires,
  - Intégration d'autres paradigmes : fonctionnel, concurrent, ...
  - génie logiciel orienté modèles (voir cours Ingénierie Logicielle - M1)

Les langages à objets (2) : caractéristiques

- avec structuration en classes
  - typés dynamiquement (SmallTalk)
  - typés statiquement
    - sous-typage nominal (C++, Java, C#, Scala)
    - sous-typage structurel (Objective Caml)

- sans classes
  - à base de multi-méthodes, fonctions génériques (CLOS)
  - à base de prototypes (JavaScript)

Programmation objet

Encapsulation:
Classes et Instances:
Relations : agrégat et héritage:
Redéfinition et Liaison Retardée:
Polymorphisme et Sous-Typage:

Terminologie Objet

- classe = description des données et des procédures qui les manipulent $\Rightarrow$ définit des comportements
- objet = instance d'une classe (possède tous les comportements de la classe)
- méthode = action (procédure, fonction) que l'on peut effectuer sur un objet
- envoi de message ou appel de méthode = demande d'exécution d'une méthode

Extension objet en OCaml

- extension objet $\neq$ langage objet
- langage à classes
- sans surcharge
- avec héritage multiple
- et classes paramétrées
- sous-typage $\neq$ sous-classes

Seul langage avec extension objet, statiquement typé avec inférence de types ! ! !

Classes

Déclaration d'une classe:

class [**virtual**] nom [p_1 p_2 ... p_n] =
  object [(p)]
    inherit autre_classe [p_i p_j]
    constraint typeexpr = typeexpr
    val [**mutable**] ident = expr
    initializer expr
    method [**private**] [**virtual**] nom_methode = expr
  end

Classe Point

```
1  class point (x_init,y_init) =
2  object
3    val mutable x = x_init
4    val mutable y = y_init
5    method get_x = x
6    method get_y = y
7    method moveto (a,b) =  begin x <- a; y <- b end
8    method rmoveto (dx,dy) =
9      begin x <- x + dx; y <- y + dy end
10   method to_string () =  "( "^(string_of_int  x)^
11                          ", "^(string_of_int y)^")"
12   method distance () = sqrt(float(x*x + y*y))
13 end;;
```

Qu'infère OCaml ?

2 choses:

- 1 abréviation d'un type `object`
- 1 fonction de construction à utiliser avec `new`

```
1  class point :  int * int ->
2    object
3      val mutable x : int
4      val mutable y : int
5      method distance : unit -> float
6      method get_x : int
7      method get_y : int
8      method moveto : int * int -> unit ...
9      method to_string : unit -> string   end
```

Objets et égalité

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.

```
1  # let p1 = new point (0,0);;
2  val p1 : point = <obj>
3  # let p2 = new point (3,4);;
4  val p2 : point = <obj>
5  # let p3 = new point (0,0);;
6  val p3 : point = <obj>
7  # p1 == p3;;
8  - : bool = false
9  # p1 = p3;;
10 - : bool = false
```

Envoi de messages

Un objet sait répondre à un envoi de message du nom d'une méthode de sa classe suivi des paramètres du bon type.
On utilise la notation # :

```
1  # p1#get_x;;                    - : int = 0
2  # p2#get_y;;                    - : int = 4
3  # p1#to_string();;              - : string = "( 0, 0)"
4  # p2#to_string();;              - : string = "( 3, 4)"
5  # if (p1#distance()) = (p2#distance())
6      then print_string ("c'est le hasard\n")
7      else print_string ("on pouvait parier\n");;
8  on pouvait parier
```

Type des instances

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

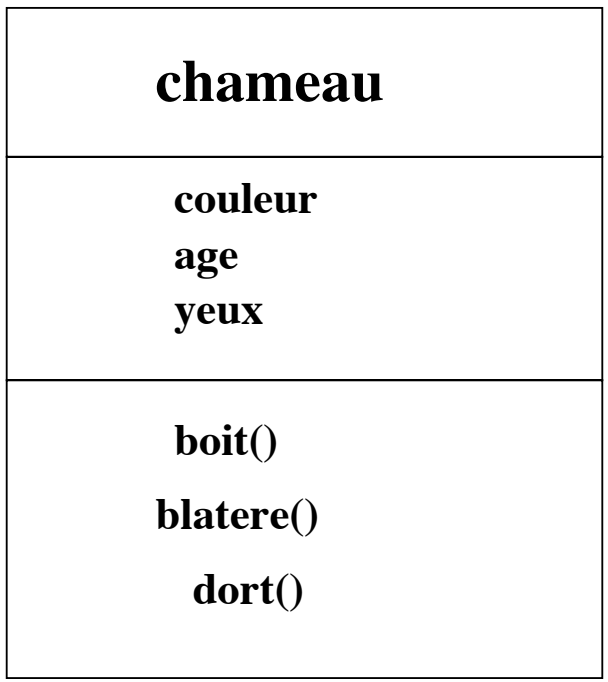
```
1 point =
2   < distance : unit -> float; get_x : int; get_y : int;
3     moveto : int * int -> unit; rmoveto : int * int -> ↩
          unit;
4     to_string : unit -> unit; >
```

correspondant aux types de ses méthodes.

Typage statique: garantie que les requêtes (appel de méthode) pourront être traitées.

Notation graphique des classes

Les classes se notent par un rectangle constitué de trois parties. Une partie portant le nom de la classe. Une autre où figure les attributs d'une instance de la classe. Enfin une dernière où sont inscrites les méthodes d'une instance de la classe.



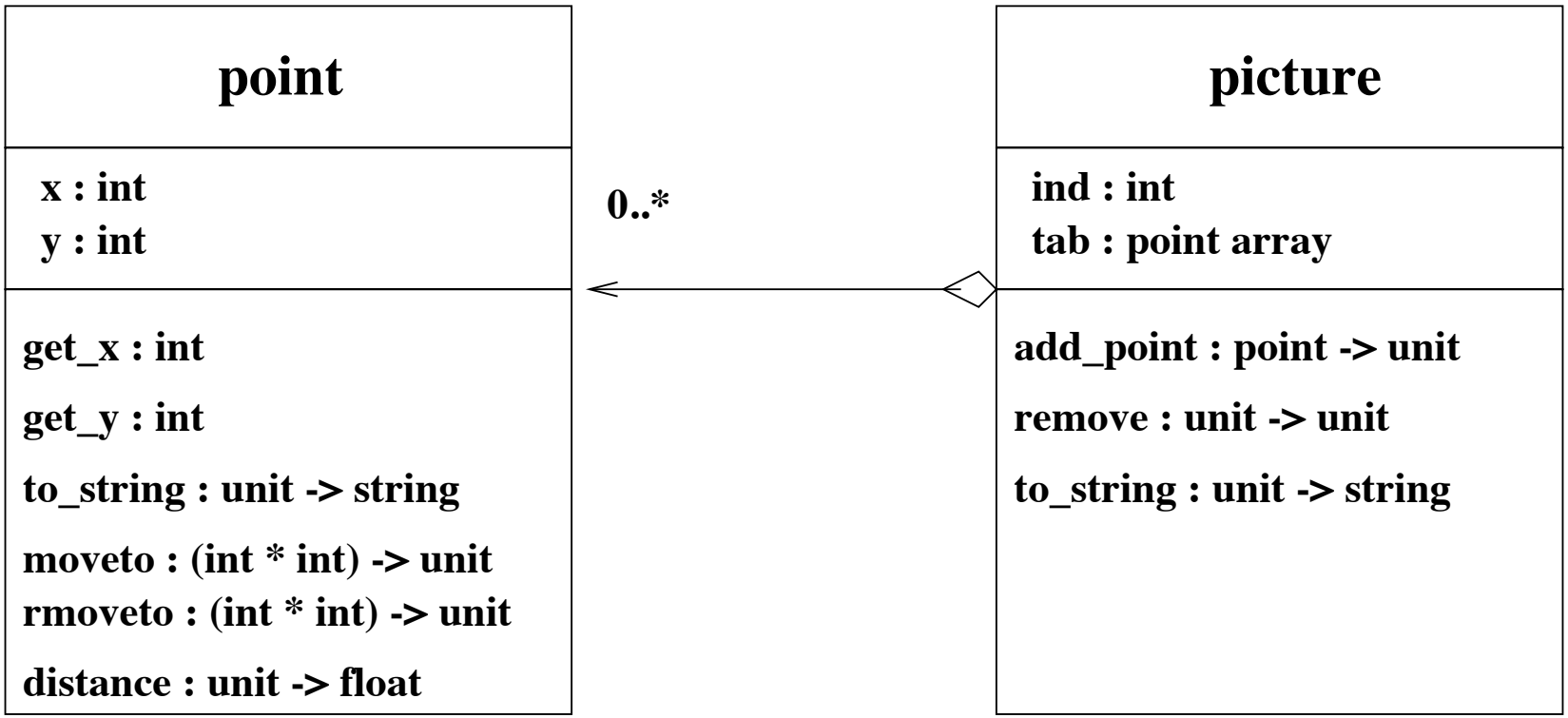
Relations entre objets

- relation d'agrégation : **Has-a**
  ex1: C_1 a un champs de C_2
  ex2: C_1 a de 0 à n champs de C_2,
- relation d'héritage : **Is-a**
  ex3: SC est sous-classe de C
  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.

Exemple d'agrégat

classe `picture`:

- possède entre 0 à *n* instances de `point`



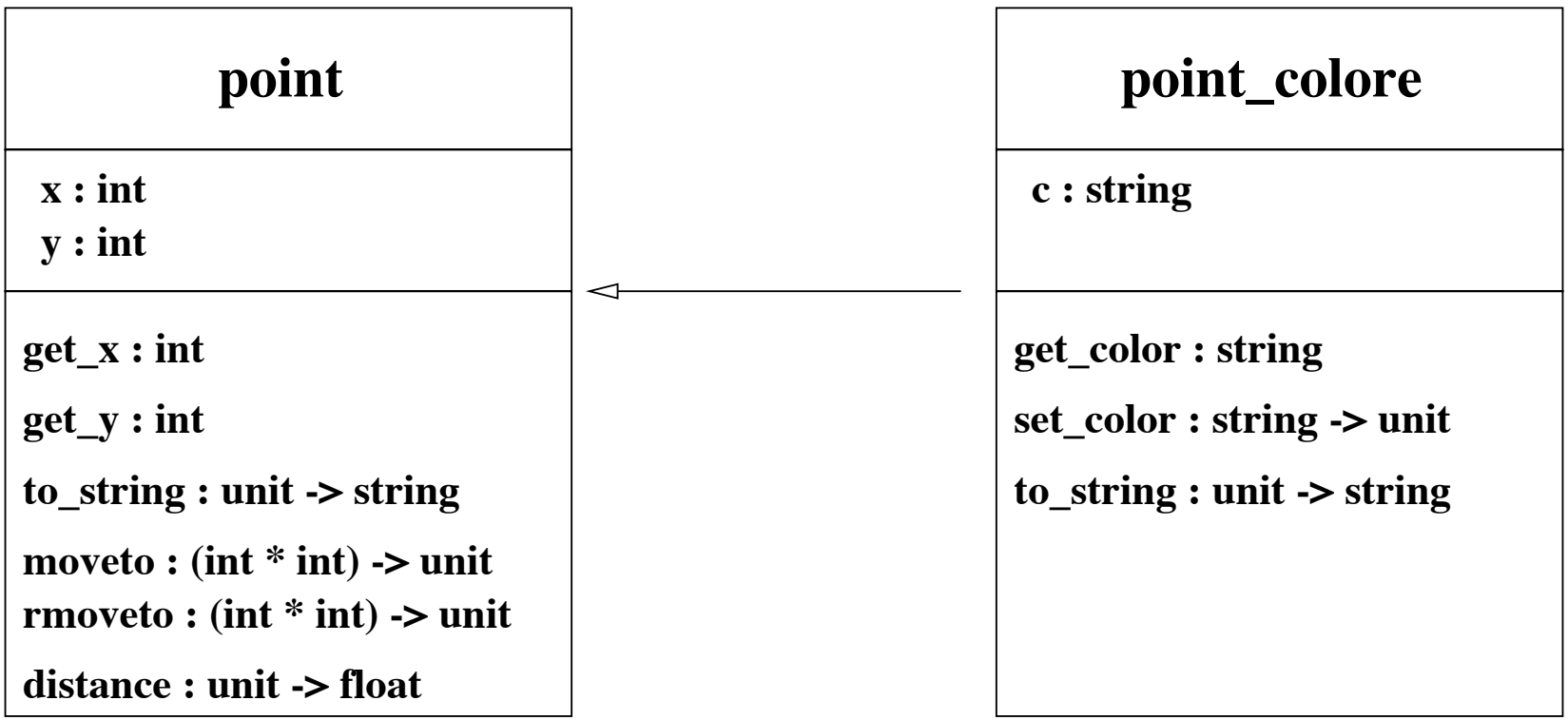
Code de la classe picture

```
1  class picture n =
2  object
3    val mutable ind = 0
4    val tab = Array.create n (new point(0,0))
5    method add p =  if (ind < n -1) then  begin
6         tab.(ind)<-p; ind <- ind + 1 end
7      else failwith ("picture.add:ind ="^(string_of_int ind)↩
           )
8    method remove () = if (ind > 0) then ind <-ind-1
9    method to_string () = let s = ref "" in
10       for i=0 to ind do s:= !s^" "^tab.(i)#to_string() ↩
             done;
11       !s
12 end;;
```

Exemple d'héritage

classe `point_colore`:

- hérite de la classe `point`



Code de la classe point_colore

```
1  class point_colore ((x_init,y_init) as paire) c =
2  object
3    inherit point paire
4    val mutable c = c
5    method get_color = c
6    method set_color nc = c <- nc
7    method to_string () =
8        "( " ^ (string_of_int x) ^
9        ", " ^ (string_of_int y) ^ ")"^
10       " de couleur " ^ c
11 end;;
```

Référencement : self et super

```
1  class point_colore p c  =
2  object(self)
3    inherit point p as super
4    val c = c
5    method get_color = c
6    method to_string () =
7    begin
8        super#to_string() ^" de couleur "^ self#get_color
9    end
10 end;;
```

Les noms sont libres, mais on utilisera `this` ou `self` pour soi-même et `super` pour la classe ancêtre.

Liaison retardée

- liaison “tardive” ou “retardée” (ou liaison “dynamique”) :

  détermination à l'exécution de la méthode à utiliser lors de l'envoi d'un message

- liaison “précoce” (ou liaison “statique”) :

  effectue cette résolution à la compilation

OCaml implante la liaison retardée ! ! !

Recherche d'une méthode

Le typage statique garantit que l'envoi d'un message correspondra bien à l'appel d'une méthode de même nom et de même signature.

Exemple de liaison retardée

On modifie la méthode `distance` de la classe `point` :

```
1 method distance () =
2    sqrt(float(self#get_x*self#get_x) +.
3         float(self#get_y*self#get_y) +. )
```

et on redéfinit la méthode `get_x` de `point_colore` :

```
1   method get_x = x * 2
```

Alors

```
1 (new point_colore (2,3) "bleu")#distance()
```

retourne la valeur 5.0 $\neq \sqrt{4+9}$

$\Rightarrow$ permettant de modifier le comportement de méthodes héritées.

Initialisation

initialiseur: méthode particulière déclenchée immédiatement après la construction de l'objet.

```
1  class point (x_init,y_init) =
2  object  ...
3  initializer print_string "Creation d'un point";
4                print_newline(); flush stdout
5  end;;
6  class point_colore p c =
7  object
8    inherit point p ...
9    initializer print_string "Creation d'un point colore";
10               print_newline(); flush stdout
11 end;;
```

Trace d'initialiseurs

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

```
1
2  # let p = new point;;
3  val p : int * int -> point = <fun>
4  # let p = new point (3,4);;
5  Creation d'un point
6  val p : point = <obj>
7  # let pc = new point_colore (3,4) "blanc";;
8  Creation d'un point
9  Creation d'un point colore
10 val pc : point_colore = <obj>
```

Visibilité

Une méthode peut être déclarée `private` :

- n'apparait pas dans l'interface de la classe (donc dans le type d'un objet de cette classe)
- s'hérite et donc utilisable dans la sous-hiérarchie

```
1  class point (x_init,y_init) =
2  ...
3    method private rmoveto (dx,dy) =
4    begin x <- x + dx;
5          y <- self#get_y + dy
6    end
7    method step1 = self#rmoveto(1,1)
8  ...
9  end;;
```

Exemple de méthodes privées

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

```
1  # let p = new point (2,3);;
2  Creation d'un point
3  val p : point = <obj>
4  # p#to_string();;
5  - : string = "( 2, 3)"
6  # p#step1;;
7  - : unit = ()
8  # p#to_string();;
9  - : string = "(3, 4)"
10 # p#rmoveto(1,1);;
11 This expression has type point. It has no method rmoveto
```

Classes et méthodes abstraites

classe abstraite: classe dont certaines méthodes ne possèdent pas de corps.

- ces méthodes sont dites *abstraites* ;
- utilisation du mot clé `virtual`.

Si une sous-classe, d'une classe abstraite, redéfinit toutes les méthodes abstraite de l'ancêtre, alors elle devient concrète, sinon elle reste abstraite.

Exemple d'une classe abstraite

```
1  class virtual graphical_object () =
2  object(self)
3    method virtual to_string : unit -> string
4    method display () = print_string (self#to_string())
5  end;;
```

L'interface calculée est la suivante :

```
1  class virtual graphical_object :
2    unit ->
3    object
4      method display : unit -> unit
5      method virtual to_string : unit -> string
6    end
```

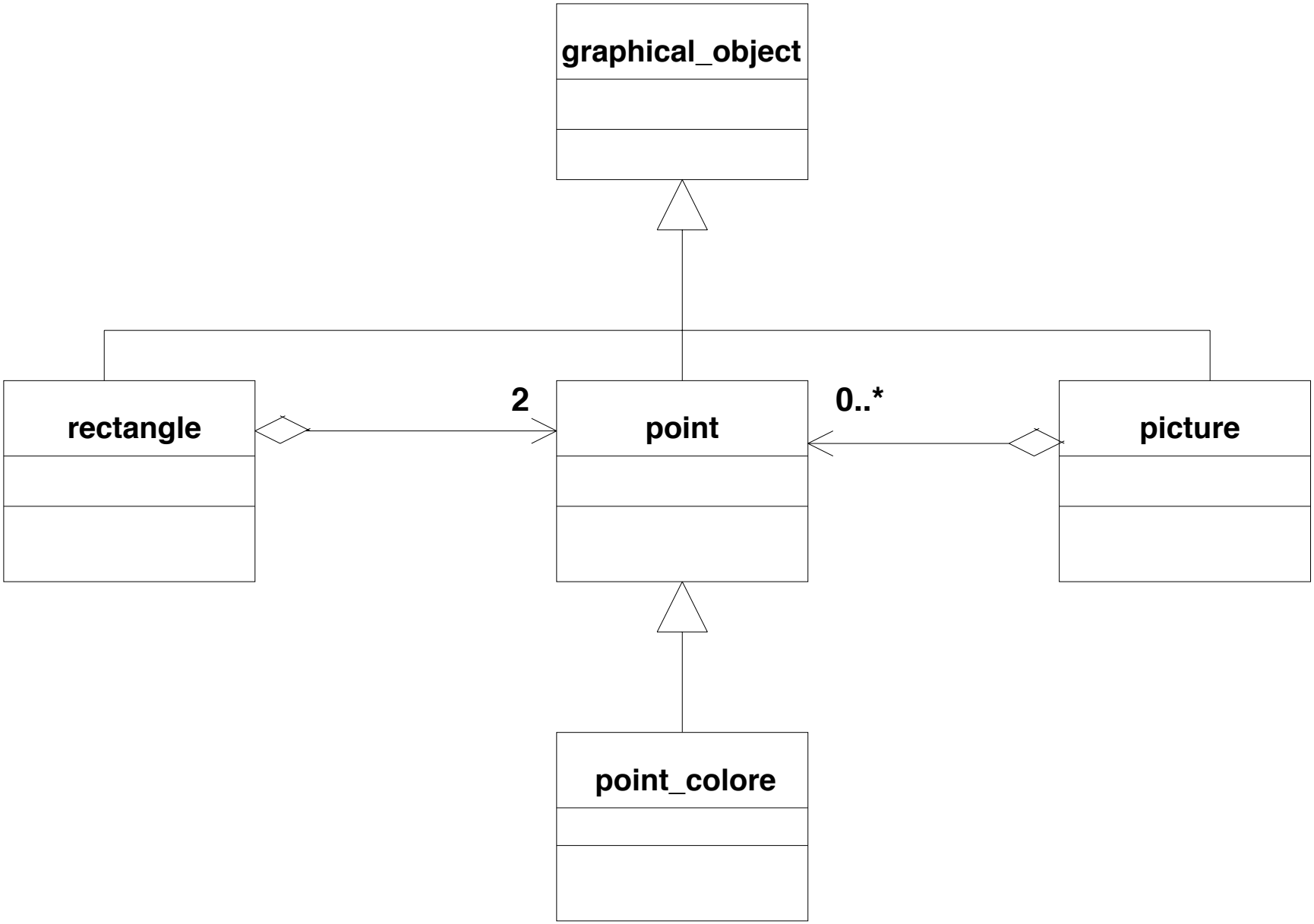
Classe rectangle

```
1  class rectangle (p1,p2) =
2  object
3    inherit graphical_object ()
4    val mutable llc = (p1 : point)
5    val mutable ruc = (p2 : point)
6    method to_string()="("^p1#to_string()^","^
7                        p2#to_string()^")"
8  end;;
```

d'interface

```
1  class rectangle :
2    point * point ->  object
3      val mutable llc : point
4      val mutable ruc : point
5      method to_string : unit -> string  end
```

Diagramme des relations de classes



Objets et types

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

```
1 point =<distance: unit -> float; get_x: int; get_y: int;
2          moveto: int * int -> unit; rmoveto: int*int-> unit↩
               ;
3          to_string: unit -> string >
```

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

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

Types ouverts

type ouvert: est représenté par la notation `< ..>`,
pour passer d'un type objet fermé à un type objet ouvert, on utilisera alors la notation `#type_obj` comme dans l'exemple suivant :

```
1 # let g (x : #point) = x#amess;;
2 val g :
3 <amess: 'a; distance: unit -> float; get_x: int; get_y: ↩
      int;
4  moveto: int * int -> unit; to_string: unit -> string;
5  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`.

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.
Intérêt: augmenter la réutilisabilité des classes.

Exemple d'héritage multiple

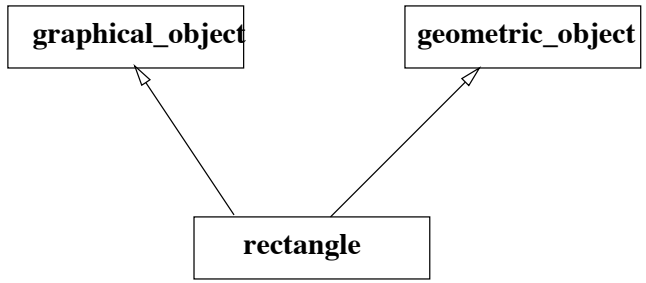
On définit la classe abstraite `geometric_object` qui déclare deux méthodes `compute_area` et `compute_circ` pour le calcul de la surface et du périmètre.

```
1 class virtual geometric_object () =
2 object
3   method virtual compute_area : unit -> float
4   method virtual compute_circ : unit -> float
5 end;;
```

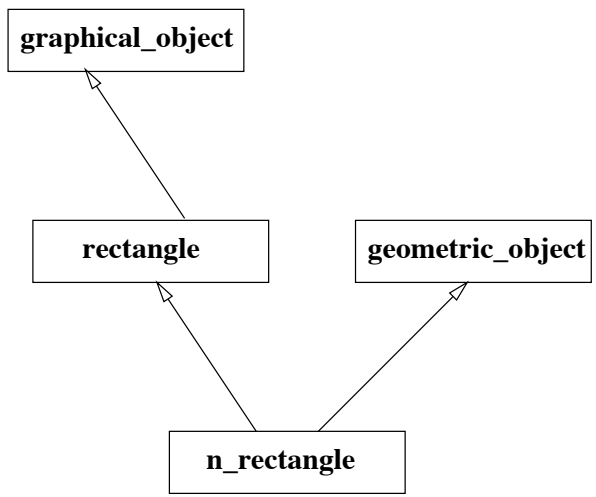
Nouvelle classe `rectangle`

```
1  class rectangle (p1,p2) =
2  object
3    inherit graphical_object ()
4    inherit geometric_object ()
5    val mutable llc = (p1 : point)
6    val mutable ruc = (p2 : point)
7    method to_string()="("^p1#to_string()^","^
8                        p2#to_string()^")"
9    method compute_area() = abs(ruc#get_x - llc#get_x) *
10                           abs(ruc#get_t - llc#get_y)
11   method compute_circ() = (abs(ruc#get_x - llc#get_x) +
12                           abs(ruc#get_t - llc#get_y)) * 2
13 end;;
```

Modélisation de l'héritage multiple (1)



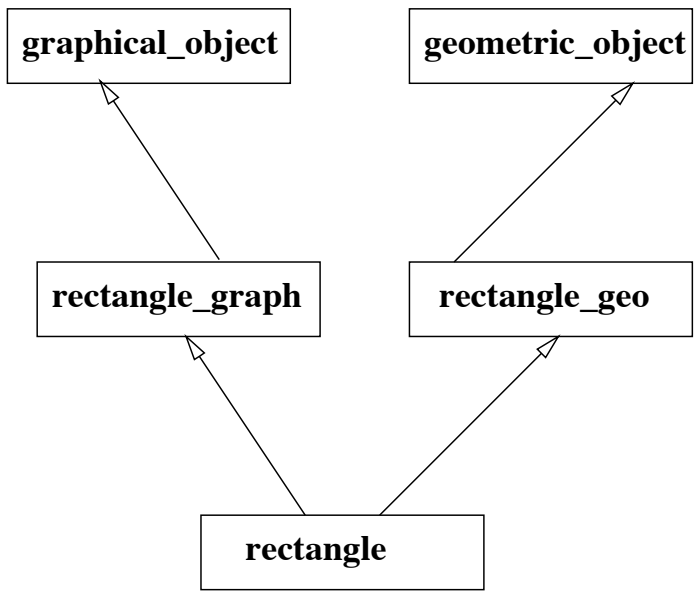
Autres modélisations:



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

Modélisation de l'héritage multiple (2)

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 :



Classes paramétrées

Utilisation: du polymorphisme paramétrique dans les classes
Intérêt: augmente la généricité du code

```
1  class ['a,'b] pair (a:'a) (b:'b) =
2    object
3      val x = a
4      val y = b
5      method fst = x
6      method snd = y
7    end;;
8  # let v = new pair 3 true;;
9  val v : (int, bool) pair = <obj>
```

Classe paramétrée `Pile`

```
1  class ['a] pile ((x:'a),n) =
2  object(self)
3   val mutable ind = 0
4   val tab = Array.create n x
5   method is_empty () = if ind = 0 then true else false
6   method private is_full () =
7     if ind = n+1 then true else false
8   method pop() =
9     if self#is_empty() then failwith "pile vide"
10    else ind <- ind -1 ; tab.(ind)
11  method push y =
12    if self#is_full() then failwith "pile pleine"
13    else tab.(ind) <- y; ind <- ind + 1
14 end;;
```

Utilisation de la classe `pile`

```
# 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 = ()
```

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 :

```
1  class virtual ['a] liste () =
2  object
3    method virtual empty : unit -> bool
4    method virtual cons  : 'a -> unit
5    method virtual head :  'a
6    method virtual tail : 'a liste
7    method virtual display : unit -> unit
8  end;;
```

Sous-classes de `liste`

```
1  class ['a] cons (v,l) =
2  object (self)
3    inherit ['a] liste ()
4    val mutable car = (v:'a)
5    val mutable cdr = (l:'a liste)
6    method empty () = false
7    method cons x = cdr<-new cons(car,cdr); car<-x
8    method head = car
9    method tail = cdr
10   method display () =
11      car#print(); print_string "  ::";
12      self#tail#display()
13 end;;
```

Contrainte de type implicite

contrainte inférée

```
1  class ['a] cons :
2    'a * 'a liste ->
3    object
4      constraint 'a = < print : unit -> 'b; .. >
5      val mutable car : 'a
6      val mutable cdr : 'a liste
7      method cons : 'a -> unit
8      method display : unit -> unit
9      method empty : unit -> bool
10     method head : 'a
11     method tail : 'a liste
12   end
```

Exemple : listes en objet

```
 1  exception ListeVide;;
 2  class  ['a] nil () =
 3  object (self)
 4    inherit ['a] liste ()
 5    val nil = ()
 6    method empty () = true
 7    method cons (x:'a) = failwith "bad argument"
 8    method head = raise ListeVide
 9    method tail = raise ListeVide
10    method display () = print_string "[]"
11  end;;
```

Contrainte de type explicite

```
1  class virtual printable () =
2  object
3    method virtual print : unit -> unit
4  end;;
5
6  class ['a] cons (v,l) =
7  object (self)
8    inherit ['a] liste ()
9    constraint 'a = #printable
10   val mutable car = (v:'a)
11   val mutable cdr = (l:'a liste)
12   method empty () = false
13   method cons x = cdr<-new cons(car,cdr); car<-x
14   method head = car
15   method tail = cdr
16   method display () =
17      car#print(); print_string "  ::";
18      self#tail#display()
19 end;;
```

Utilisation des listes

On définit une classe `integer` possèdant une méthode `print`

```
1  class integer i = object
2    val v = i
3    method get = v
4    method print () = print_int v
5  end;;
```

La construction de liste est la suivante :

```
1  # let i1 = new integer 1;;
2  val i1 : integer = <obj>
3  # let i2 = new integer 2;;
4  val i2 : integer = <obj>
5  # let n = new nil ();;
6  val n : '_a nil = <obj>
7  # let l = new cons (i1,n);;
8  val l : integer liste = <obj>
9  # l#display();;
10                 1  ::[] - : unit = ()
```

Sous-typage (1)

sous-typage: est une relation entre deux types 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 $i \in \{1, \ldots, n\}$.

subsomption: est la possibilité pour un objet d'un certain sous-type d'être considéré et utilisé comme un objet d'un sur-type au sens de la relation de sous-typage.

sous-typage (2)

Notation: 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.

La relation de sous-typage, combinée avec la liaison tardive, introduit une nouvelle forme de polymorphisme : le polymorphisme d'**inclusion**.

Sous-typage et polymorphisme d'inclusion

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>
```

Invocation: de la méthode `to_string`

```
# p#to_string();;
- : string = "( 4, 5)"
# pc#to_string();;
- : string = "( 4, 5) de couleur blanc"
# np#to_string();;
- : string = "( 4, 5) de couleur blanc"
```

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

Exemple

Construction: d'une liste de points

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

Cela vient de la liaison tardive (choix de la méthode à utiliser à l'exécution).

Sous-typage $\neq$ héritage

2 arguments:

- on peut être sous-type sans héritage

  il est possible de forcer un type classe dans un autre type classe sans que le premier corresponde à un descendant du deuxième

- on peut hériter sans être sous-type

  cela arrive quand une des méthodes de la classe ancêtre a un paramètre du type de la classe

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 fonctions: Si $f : \sigma \to \tau$ dans C, $a : \sigma'$ dans C et
$\sigma' \leq \sigma$ dans C
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 (1)

Si on définit les classes suivantes :

```
1  class a =
2   ...
3    method f : t1 -> t2
4   ...
5  end;;
6  class b =
7   ...
8    method f : t3 -> t4
9   ...
10 end;;
```

Si on veut montrer que $b \leq a$ alors il faut vérifier $(t_3 \to t_4) \leq (t_1 \to t_2)$.

sous-typage de types fonctionnels (2)

Pour distinguer les deux méthodes f on les nomme : f_a et f_b.
Soient $t_1 \to t_2$ et $t_3 \to t_4$ deux types fonctionnels, ils sont en relation de sous-typage :
$(t_3 \to t_4) \leq (t_1 \to t_2)$

si et seulement si :

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

Justification (1)

Soient les 2 fonctions suivantes bien typées :

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

avec

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

- **[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 :

$$type\_res(f_b) \leq type\_res(f_a) \Rightarrow t_4 \leq t_2$$

Justification (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.

Exemples

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

$$eq_{point} : point \to bool \quad eq_{point\_colore} : point\_colore \to bool$$

et on s'aperçoit alors que pour que

$$point\_colore \leq point$$

il faudrait que

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

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

$$point \leq point\_colore$$

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.

Copie d'objets: On utilise pour cela l'annotation `{< ... >}` qui retourne une copie de l'objet (self) dans lequel les valeurs de certains champs de données sont changées.
La fonction `Oo.copy` retourne une copie d'un objet. Son type est le suivant :

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

Exemple en style fonctionnel

```
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 to_string () =  "( "^(string_of_int  x)^
                         ", "^(string_of_int y)^")" end;;
```

```
class point :
  int * int ->
  object ('a)
    method moveto : int * int -> 'a
    method rmoveto : int * int -> 'a
    method to_string : unit -> string
  end
```

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

Style fonctionnel et typage (1)

```
1  class point_c ((x_init,y_init) as p) c = object (self)
2    inherit point p as super
3    val c = c
4    method get_c = c
5    method to_string() = super#to_string() ^ " " ^ self#↩
         get_c
6  end
```

```
1  class point_c :
2    int * int ->
3    string ->
4    object ('a)
5      method get_c : string
6      method moveto : int * int -> 'a
7      method rmoveto : int * int -> 'a
8      method to_string : unit -> string
9    end
```

```
1  # let pc = new point_c (2,3) "B";;
2  val pc : point_c = <obj>
3  # (pc#rmoveto(10,10))#to_string();;
4  - : string = "( 12, 13) B"
5  # pc#to_string();;
6  - : string = "( 2, 3) B"
```

Style fonctionnel et typage (2)

```
1  class point_p (x_init,y_init) = object
2    val x = x_init
3    val y = y_init
4    method moveto (a,b) = new point_p (a,b)
5    method rmoveto (dx,dy) = new point_p (x+dx,y+dy)
6    method to_string () =  "( "^(string_of_int  x)^
7                           ", "^(string_of_int y)^")" end;;
```

```
1  class point_p_c ((x_init,y_init) as p) c = object (self)
2    inherit point_p p as super
3    val c = c
4    method get_c = c
5    method to_string() = super#to_string() ^ " " ^ self#↩
         get_c
6  end
```

```
1  # let pc = new point_p_c (2,3) "B";;
2  val pc : point_p_c = <obj>
3  # (pc#rmoveto(10,10))#to_string();;
4  - : string = "( 12, 13)"                    (* couleur ? *)
5  # pc#to_string();;
6  - : string = "( 2, 3) B"
```

Interfaces (1)

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

```
1  class type interf_point  =
2  object
3    method get_x : int
4    method get_y : int
5    method moveto : (int * int ) -> unit
6    method rmoveto : (int * int )  -> unit
7    method print : unit -> unit
8    method distance : unit -> float
9  end;;
```

Interfaces (2)

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

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

Une interface ne masque que les variables d'instance et les méthodes privés.

Intérêt: construire des interfaces de classes sans avoir à définir la classe pour les modules

Retour sur les classes

Déclarations de classes

class nom = function p1 -> ... -> function pn -> object end ; ;

avec possibilité de déclarations locales :

```
1  class titi =
2    let x =  ref 0 in fun y ->
3      object(self)
4        val mutable z = 0
5        val nom = (y : string)
6        method gensym () = z<-z+1;
7          y^"_"^(string_of_int !x)^"_"^(string_of_int z)
8        initializer incr x
9      end;;
```

Exemple d'exécution

```
1  # let a = new titi "R";;
2  val a : titi = <obj>
3  #  a#gensym();;
4  - : string = "R_1_1"
5  #  a#gensym();;
6  - : string = "R_1_2"
7  # let b = new titi "T";;
8  val b : titi = <obj>
9  #  b#gensym();;
10 - : string = "T_2_1"
11 #  b#gensym();;
12 - : string = "T_2_2"
13 # a#gensym();;
14 - : string = "R_2_3"
```

Méthodes polymorphes (1)

```
1  exception Empty
2  class oqueue () =
3    object(self)
4      val mutable q = []
5      method enq x =  q <-  q @ [x]
6      method deq () = match q with [] -> raise Empty
7                                 |  h::r -> q <- r ; h
8      method reset () = q <- []
9      method fold f accu = List.fold_left f accu q
10   end;;
```

```
1  File ``oqueue.ml'', line 3, characters 5-260:
2  Some type variables are unbound in this type:
3    class oqueue :
4      unit ->
5      object
6        val mutable q : 'a list
7        method deq : unit -> 'a
8        method enq : 'a -> unit
9        method fold : ('b -> 'a -> 'b) -> 'b -> 'b
10       method reset : unit -> unit
11     end
12 The method deq has type unit -> 'a where 'a is unbound
```

Méthodes polymorphes (2)

```
1  exception Empty
2  class ['a, 'b] oqueue2 () =
3    object(self)
4      val mutable q = ([] : 'a list)
5      method enq x =  q <-  q @ [x]
6      method deq () = match q with
7          [] -> raise Empty
8      |  h::r -> q <- r ; h
9      method reset () = q <- []
10     method fold f (accu : 'b) = List.fold_left f accu q
11   end;;
```

```
1  class ['a, 'b] oqueue2 :
2    unit ->
3    object
4      val mutable q : 'a list
5      method deq : unit -> 'a
6      method enq : 'a -> unit
7      method fold : ('b -> 'a -> 'b) -> 'b -> 'b
8      method reset : unit -> unit
9    end
```

Méthodes polymorphes (3)

```
# let oq = new oqueue2();;
val oq : ('_a, '_b) oqueue2 = <obj>
# oq#enq ``Salut'';;
- : unit = ()
# oq;;
- : (string, '_a) oqueue2 = <obj>
# oq#enq ``bye'';;
- : unit = ()
# oq;;
- : (string, '_a) oqueue2 = <obj>
# oq#fold (fun x y -> x + (String.length y) ) 0;;
- : int = 8
# oq;;
- : (string, int) oqueue2 = <obj>
#  oq#fold (fun x y -> x || (y = ``Fin'')) false;;
This expression has type int but is here used with type
    bool
```

Méthodes polymorphes (4)

- Si le polymorphisme d'une méthode est indépendant de variables de type de la classe de définition, alors il n'est pas dangereux de le lier localement.
- liaison explicite en indiquant le type et les variables quantifiés :

```
1 method nom : 'a 'b. ('a -> 'b -> 'a) = expr
```

Méthodes polymorphes (5)

```
exception Empty
class ['a] oqueue3 () =
  object(self)
    val mutable q = ([] : 'a list)
    method enq x =  q <-  q @ [x]
    method deq () = match q with
        [] -> raise Empty
    |  h::r -> q <- r ; h
    method reset () = q <- []
    method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =  fun ↩
        f accu ->
       List.fold_left f accu q
  end;;
```

```
class ['a] oqueue3 :
  unit ->
  object
    val mutable q : 'a list
    method deq : unit -> 'a
    method enq : 'a -> unit
    method fold : ('b -> 'a -> 'b) -> 'b -> 'b
    method reset : unit -> unit
  end
```

Méthodes polymorphes (6)

```
 1 # let oq = new oqueue3 ();;
 2 val oq : '_a oqueue3 = <obj>
 3 # oq#enq ``Salut'';;
 4 - : unit = ()
 5 # oq;;
 6 - : string oqueue3 = <obj>
 7 # oq#enq ``bye'';;
 8 - : unit = ()
 9 # oq#fold (fun x y -> x + (String.length y) ) 0;;
10 - : int = 8
11 # oq;;
12 - : string oqueue3 = <obj>
13 # oq#fold (fun x y -> x || (y = ``Fin'')) false;;
14 - : bool = false
```

Objets immédiats (1)

- création d'objet sans être instance de classe :

```
# let p =
  object
    val mutable x = 0
    val mutable y =0
    method get_x = x
    method get_y = y
    method rmoveto dx dy = x <- x + dx; y <- y + dy
  end;;
val p : < get_x : int; get_y : int; rmoveto : int -> ↩
    int -> unit > = <obj>
# p#rmoveto 1 2;;
- : unit = ()
# p#get_y;;
- : int = 2
```

Objets immédiats (2)

- manipulation du type de self :

```
# let p3 = object(self:'a)
    val mutable x = 0
    val mutable y =0
    method get_x = x
    method get_y = y
    method rmoveto dx dy = x <- x + dx; y <- y + dy
    method eq (z:'a) = x = (z#get_x)
    method id () = self
  end;;
val p3 :
  < eq : 'a -> bool; get_x : int; get_y : int; id : ↩
      unit -> 'a;
    rmoveto : int -> int -> unit >
  as 'a = <obj>
```

Objets immédiats (3)

- variables de type classiques :

```
# let r2  =
  object(self)
    val mutable q = ([] : 'a list)
    method enq x =  q <-  q @ [x]
  end ;;
val r2 : < enq : '_a -> unit > = <obj>
# let r4 z =
  object(self)
    val mutable q = z
    method enq x =  q <-  q @ [x]
  end ;;
val r4 : 'a list -> < enq : 'a -> unit > = <fun>
```

Objets immédiats (4)

- avantages
  - utilisable dans une fonction, ou un foncteur
  - moins de contraintes de types
- désavantages
  - pas d'héritage
  - type anonyme

Rappel : application non généralisée (1)

```
1  # let id = function x -> x;;
2  val id : 'a -> 'a = <fun>
3
4  # let i = id id;;
5  val i : '_a -> '_a = <fun>
6
7  # i 3;;
8  - : int = 3
9
10 # i;;
11 - : int -> int = <fun>
12
13 # i true;;
14 This expression has type bool
15 but is here used with type int
```

Rappel : application non généralisée (2)

Solution: ajouter un paramètre

```
1  # let j y = id id y;;
2  val j : 'a -> 'a = <fun>
3
4  # j 3;;
5  - : int = 3
6
7  # j true;;
8  - : bool = true
```

variables de type faibles et objets (1)

```
1  class p x_i y_i =
2    object
3      val mutable x = x_i
4      val mutable y = y_i
5      method rmoveto dx dy = x <- x + dx; y <- y + dy
6    end;;
7
8  class ['a,'b] pair (x0:'a) (y0:'b) =
9     object
10      val x = x0
11      val y = y0
12      method fst = x
13      method snd = y
14    end ;;
```

variables de type faibles et objets (2)

```
1  # let jolitype x ( y : ('a, 'a) #pair) =
2      if x = y#fst then y else y;;
3  val jolitype : 'a -> (('a, 'a) #pair as 'b) -> 'b = <fun>
4
5  # let g = jolitype 3;;
6  val g : ((int, int) _#pair as 'a) -> 'a = <fun>
7
8  class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
9     object
10      inherit ['a,'b] pair x0 y0
11      method get1 z = if x = z then y else raise Not_found
12      method get2 z = if y = z then x else raise Not_found
13    end;;
```

variables de type faibles et objets (3)

```
1 # let h = g (new acc_pair 8 6);;
2 val h : < fst : int; get1 : int -> int; get2 : int -> int;↩
        snd : int > =
3   <obj>
4 # g;;
5 - : < fst : int; get1 : int -> int; get2 : int -> int; snd↩
        : int > ->
6     < fst : int; get1 : int -> int; get2 : int -> int; snd↩
            : int >
7 = <fun>
```