Modules et objets pour la programmation générique en Objective Caml

—◦—

Pascal Manoury

– Mai 2008 –

—◦—

CIRM Lumigny

Le contexte

Objective Caml

- noyau pleinement fonctionnel (λ-calcul)

+ types algébriques (filtrage), n-uplets, enregistrements

+ structures impératives, valeurs modifiables

+ langage de modules, foncteurs

+ extension objet

Et tout ça, avec

++ ***inférence de type statique avec polymorphisme paramétrique***

Thème de cette journée

Généricité

Selon deux axes

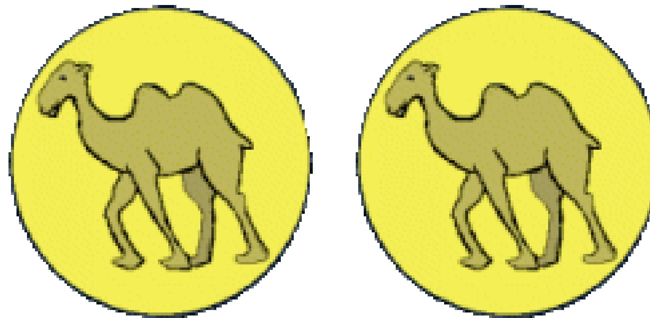
- programmation par modules
- programmation par objets

Références

- Développement d'applications avec Objective Caml, E. Chailloux, P. Manoury, B. Pagano, O'Reilly 2000 – épuisé:(*mais disponible en ligne*)
- Programmation fonctionnelle, générique et objet. Une introduction avec le langage OCaml, P. Narbel, Vuibert 2005

Première partie

Les modules et leur langage
en Objective Caml



Modules en Objective Caml

Structuration logicielle

- ***physique:*** unité de compilation (fichiers)
- ***logique:*** syntaxe explicite + abstraction fonctionnelle

Deux composants des modules

- ***signature***: suite de déclarations
  types, exceptions, valeurs, modules, etc.
- ***implantation***: suite de définitions
  tout ce qui est requis par la signature, au moins

Analogie fonctionnelle

signature $\approx$ type

Structures

Syntaxe: **module** Id = **struct** ... **end**

Définition d'une implantation

```
# module M =
  struct
    type t = int * int * int
    let make d m y = d,m,y
  end
;;
module M :
  sig type t = int * int * int
      val make : 'a -> 'b -> 'c -> 'a * 'b * 'c end
```

$\Rightarrow$ Signature inférée: le type le plus général

Structures (suite)

Usage

```
# let d = M.make 08 05 08 ;;
val d : int * int * int = (8, 5, 8)
```

- Notation pointée
- Le type instancié

Autre usage (polymorphisme)

```
# let d = M.make 08 "mai" 2008 ;;
val d : int * string * int = (08, "mai", 2008)
```

⇒ incohérence type attendu / types obtenus

Signature

Syntaxe: **module type** Id = **sig** ... **end**

Spécification

```
# module type S =
  sig
    type t = int * int * int
    val make : int -> int -> int -> t
  end
```

« `make` doit fabriquer des valeurs du type attendu »

Idée un ensemble de valeurs et leurs traitements

Signature (bis)

Usage contrainte de type

```
# module MS = (M:S) ;;
module MS : S
# MS.make;;
- : int -> int -> int -> MS.t = <fun>
```

⇒ un nouveau type `MS.t`

⇒ avec son constructeur `MS.make`

Contrôle de type

```
# MS.make 1 2 3;;
- : MS.t = (1, 2, 3)
# MS.make 8 "mai" 2008;;
This expression has type string but is here used with type int
```

Application: signature

Un module pour les dates

Spécification des types attendus

```
# module type DATE =
  sig
    type t = int * int * int
    val mmax : int -> int -> int
    val make : int -> int -> int -> t
    val get_day : t -> int
    val get_month : t -> int
    val get_year : t -> int
  end
```

Application: implantation

Dates: contrôler la vraissemblance des valeurs[1]

```
module M = struct
    type t = int * int * int
    let mmax m y =
      match m with
        1 | 3 | 5 | 7 | 8 | 10 | 12 -> 31
      | 2 ->  if (y mod 4 = 0) then 29 else 28
      | _ -> 30
    let make d m y =
      if (m < 1) || (12 < m) || (d < 1) || ((mmax m y) < m) then
        raise (Invalid_argument "make");
      d, m, y
    let get_day (d,m,y) = d
    let get_month (d,m,y) = m
    let get_year (d,m,y) = y
  end
```

[1] au moins partiellement

Application: application

Création du module par contrainte de signature

```
# module Date = (M:DATE) ;;
module Date : DATE
```

Pourquoi avoir fait «signature» puis «implantation» ?

1. on dit ce que l'on veut (signature)
2. on fait ce que l'on veut (structure)
3. on[1] vérifie qu'on ne s'est pas planté (contrainte)

---

(1) c'est-à-dire, l'inférence de type du compilateur

Application: utilisation

Contrôle de vraissemblance: opérationnel

⇒ les valeurs ont le type attendu

```
# Date.make 08 05 08;;
- : Date.t = (8, 5, 8)
```

⇒ les valeurs incorrectes ne sont pas construites

```
# Date.make 12 34 56;;
Exception: Invalid_argument "make".
```

Mais: contrôle non maîtrisé

```
#  M.get_month (23,45,67);;
- : int = 45
```

Fonctions du module applicables à (presque) n'importe quoi.

Application: analyse

Le problème: on peut construire des valeurs avec d'autres moyens que ceux offerts par le module

Origine du problème:

l'implantation du type `Date.t` est ***publique***

⇒ `Date.t` est un ***alias*** pour `int * int * int`

⇒ l'inférence de type ne les distingue pas, donc les accepte

Solution: ***masquer*** l'implantation du type

• Type abstrait de données •

Application: type abstrait

Nouvelle signature

```
# module type DATE = sig
    type t
    val mmax : int -> int -> int
    val make : int -> int -> int -> t
    val get_day : t -> int
    val get_month : t -> int
    val get_year : t -> int
  end
```

Nouveau module

```
# module Date = (M:DATE) ;;
module Date : DATE
```

Contrôle maîtrisé

```
#  Date.get_month (23,45,67);;
This expression has type int * int * int but is here used with type Date.t
```

Modules et unités de compilation

Sur la ligne de commande du compilateur

· signature = fichier `.mli`

· implantation = fichier `.ml`

· contrainte de signature = même nom de fichier

Dans les fichiers sources l'extension fait le travail

- pas de `module .. = struct .. end`;
- ni de `module type .. = sig .. end`.

Les fichiers

date.ml	date.mli

`type t = int * int * int`
`let mmax m y =`
`  match m with`
`    1 | 3 | 5 | 7 | 8 | 10 | 12`
`   -> 31`
`  | 2`
`   ->  if (y mod 4 = 0) then 29`
`       else 28`
`  | _ -> 30`
`let make d m y =`
`  if (m < 1) || (12 < m) ||`
`     (d < 1) || ((mmax m y) < m)`
`  then`
`    raise (Invalid_argument "make");`
`  d, m, y`
`let get_day (d,m,y) = d`
`let get_month (d,m,y) = m`
`let get_year (d,m,y) = y`
	`type t`
	`val mmax : int -> int -> int`
	`val make : int -> int -> int -> t`
	`val get_day : t -> int`
	`val get_month : t -> int`
	`val get_year : t -> int`

Compilation et test

```
eleph@argent:~/Luminy$
 ocamlc -c date.mli date.ml
eleph@argent:~/Luminy$
 ocaml
        Objective Caml version 3.09.2

# #load "date.cmo";;
# Date.make;;
- : int -> int -> int -> Date.t = <fun>
# let d = Date.make 08 05 08;;
val d : Date.t = <abstr>
# Date.get_year d;;
- : int = 8
# match d with _,_,y -> y ;;
This pattern matches values of type 'a * 'b * 'c
but is here used to match values of type Date.t
```

Structure et signature

Leurs relations

Si `(M:S)` alors la structure `M` est une ***instance*** de la signature `S`.

Règles

- tous les éléments déclarés dans `S` sont définis dans `M`;
- les types inférés dans `M` sont égalisables[1] ou plus généraux que les types déclarés dans `S`.

---

[1] *i.e.* égaux, compte tenu des *alias*.

Structure et signature

Application

- une signature peut avoir plusieurs instances

$\Rightarrow$ un type abstrait peut avoir plusieurs implantations/représentations

```
module MR = struct
    type t = {day:int; month:int; year:int}
    let mmax m y = [...]
    let make d m y =
      if [...] then raise (Invalid_argument "make");
      {day = d; month = m; year = y}
    let get_day d = d.day
    let get_month d = d.month
    let get_year d = d.year
  end

module date = (MR:DATE)
```

Signature et signature

Leurs relations

Si `(M:S1)` et si `S2` est ***compatible*** avec `S1` alors `(M:S2)`

Règles

- tous les éléments déclarés dans `S2` le sont dans `S1`;
- les types dans `S1` sont égalisables ou plus généraux que les types dans `S2`;
- + un type concret dans `S1` peut être abstrait dans `S2`.

Signature et signature

Application

- une structure peut avoir plusieurs signatures

⇒ Plusieurs vues d'une même implantation

Exemple: implantation d'un compteur

```
# module Cpt =
  struct
    let x = ref 0
    let reset () = x := 0
    let next () = incr x; !x
  end
```

Créer deux modules n'ayant pas les mêmes droits sans toucher à l'implantation.

Application compteur

La vue de l'administrateur

Signature:

```
# module type ADM =
  sig
    val reset : unit -> unit
    val next : unit -> int
  end
```

Le module

```
# module Adm = (Cpt:ADM)
```

Nota: le compteur lui-même est invisible

```
# Adm.x;;
Unbound value Adm.x
```

Application compteur

La vue de l'utilisateur

Signature:

```
# module type USR =
  sig
    val next : unit -> int
  end
```

Le module

```
# module Usr = (Cpt:USR)
```

Utilisation

```
# Usr.next();;
- : int = 1
# Usr.reset();;
Unbound value Usr.reset
```

Modules et unités de compilation

Rappel: sur la ligne de commande du compilateur

· contrainte de signature = même nom de fichier

$\Rightarrow$ ***pas de double vue***

(de l'utilité du langage de modules)

Contrainte partage du code (et du compteur).

$\Rightarrow$ impossible de construire deux composants administrateurs et utilisateurs autonomes.

$\Rightarrow$ une unité qui contient les trois modules

$\Rightarrow$ un composant qui publie les deux modules mais pas le module commun ($\approx$ *package*)

Implantation et signature

<table>
<tr><th>pkgCpt.ml</th><th>pkgCpt.mli</th></tr>
<tr><td>
<pre>
module Cpt = struct
  let x = ref 0
  let reset () = x := 0
  let next () = incr x; !x
end
module type ADM = sig
  val reset : unit -> unit
  val next : unit -> int
end
module type USR = sig
   val next : unit -> int
end
module Adm = Cpt
module Usr = Cpt
</pre>
</td><td>
<pre>
module type ADM = sig
  val reset : unit -> unit
  val next : unit -> int
end
module Adm:ADM

module type USR = sig
  val next : unit -> int
end
module Usr:USR
</pre>
</td></tr>
</table>

```
ocamlc -c pkgCpt.mli pkgCpt.ml
```

Utilisation

```
        Objective Caml version 3.09.2

# #load"pkgCpt.cmo";;
# PkgCpt.Adm.x;;
Unbound value PkgCpt.Adm.x
# PkgCpt.Usr.reset;;
Unbound value PkgCpt.Usr.reset
# PkgCpt.Adm.next();;
- : int = 1
# PkgCpt.Usr.next();;
- : int = 2
# PkgCpt.Adm.reset();;
- : unit = ()
# PkgCpt.Usr.next();;
- : int = 1
#
```

Structure et structure

Leurs relations

L'inclusion

Syntaxe: **include** *ModName*

Applications

- extension des traitements
- rédefinition des traitements

$\approx$ relation d'héritage

Héritage par inclusion

Exemple d'ajout et rédéfinition

```
# module Cpt2 = struct
  include Cpt

  let get() = !x
  let next() = x := 2 * !x; !x
end ;;
module Cpt2 :
  sig
    val x : int ref
    val reset : unit -> unit
    val get : unit -> int
    val next : unit -> int
  end
```

Signature inférée: l'incluant connaît tout l'inclus

Inclusion et signature

Mais l'incluant ne connaît que tout ce que l'inclus publie

```
# module type ABSTCPT = sig
    val reset : unit -> unit
    val next : unit -> int
  end ;;
[...]
# module AbstCpt = (Cpt:ABSTCPT)
[...]
# module AbstCpt2 = struct
  include AbstCpt

  let get() = !x
  let next() = x := 2 * !x; !x
end ;;
Unbound value x
#
```

Inclusion et signature (suite)

Extension de signature par inclusion

```
module type ABSTCPT =
sig
  val reset : unit -> unit
  val next : unit -> int
end

module AbstCpt = (Cpt:ABSTCPT)
```

```
module type ABSTCPT2 =
sig
  include ABSTCPT
  val get : unit -> int
end

module AbstCpt2 = (Cpt2:ABSTCPT2)
```

Compatibilité descendante

```
module AbstCpt1 = (Cpt2:ABSTCPT)
```

« Qui peut le plus peut le moins »

Structure et structure

Autre relation

inclusion = utilisation

MAIS

utilisation ***d'un déjà là***

On veut prévoir avant d'avoir

⇒ module ***paramétré***

⇒ paramètre ***abstrait***

Des fonctions de modules:

• Les foncteurs •

Foncteurs définition et utilisation

Modèle fonctionnel

Abstraction

Syntaxe: **functor** (*Id* : *SIG*) -> **struct** ...**end**

Syntaxe: **module** Id_1 (Id_2 : *SIG*) = **struct** ...**end**

Application

Syntaxe: **module** $Id_1 = Id_2$ (*STRUC*)

Module paramétré

Supposons

```
module type SERIALZ =
  sig
    type t
    val to_string : t -> string
    val of_string : string -> t
  end ;;
```

Posons un module ***générique*** d'entrée/sortie

```
module StdIO (V:SERIALZ) =
  struct
    let writeln x = print_endline (V.to_string x)
    let readln () = V.of_string (read_line())
  end ;;
```

Module actualisé

Soit une instance de SERIALZ

```
module IntList =
  struct
    type t = int list
    let to_string ns =
      String.concat " " (List.map string_of_int ns)
    let of_string s =
      List.map int_of_string (Xstring.split ' ' s)
  end
```

Création du module par application

```
# module IntListStdIO = StdIO (IntList)
  sig val writeln : IntList.t -> unit val readln : unit -> IntList.t end
```

(IntList:SERIALZ) est vérifié à l'application

Abstraction de type

Supposons une sérialisation plus abstraite

```
module type SERIALZ = sig
    type t
    type z
    val to_z : t -> z
    val of_z : z -> t
  end ;;
```

Il y faut des lectures/écritures plus générales

```
module type GENIO = sig
    type t
    val output : t -> unit
    val input : unit -> t
  end
```

Abstraction et partage de type

Deux paramètres module de sérialisation et module d'entrée/sortie

Première tentative

```
# module SerialzIO (V:SERIALZ) (IO:GENIO) =
  struct
    let writeln x = IO.output (V.to_z x)
    let readln () = V.of_z (IO.input ())
  end ;;
This expression has type V.z but is here used with type IO.t
```

Types abstraits = incompatibilité de type *a priori*

⇒ forcer la compatibilité

⇒ expliciter le partage de type

Partage de type

Syntaxe: **with type** $\ldots = \ldots$

Deuxième tentative

```
# module SerialZIO (V:SERIALZ) (IO:GENIO with type t = V.z) =
  struct
    let writeln x = IO.output (V.to_z x)
    let readln () = V.of_z (IO.input ())
  end ;;
module SerialZIO :
  functor (V : SERIALZ) ->
    functor
      (IO :
        sig type t = V.z
            val output : t -> unit
            val input : unit -> t end)
      -> sig val writeln : V.t -> unit val readln : unit -> V.t end
#
```

Ça type: hum, essayons

On définit `IntListZ` à partir de `IntList`

```
module IntListZ =
  struct
    include IntList
    type z = string
    let to_z = to_string
    let of_z = of_string
  end
```

Entrée/sorties pour les chaînes

```
module StrIO =
  struct
    type t = string
    let output = print_endline
    let input = read_line
  end ;;
```

Ça marche!

```
# module IntListIO = SerialZIO (IntListZ) (StrIO) ;;
module IntListIO :
  sig val writeln : IntListZ.t -> unit
      val readln : unit -> IntListZ.t end
#IntListIO.writeln [1;2;3;4];;
1 2 3 4
- : unit = ()
# IntListIO.readln();;
6 7 8 9
- : IntListZ.t = [6; 7; 8; 9]
```

Pourquoi ?

- Contrainte: `IO.t` = `V.z` (cf nouvelle signature)
- Application 1: `V.z` ← `IntListZ.z` = `string`
- Application 2: `V.t` = `IntListZ.t` = `int list`

Bibliothèque standard

Le module `Set`

This module implements the set data structure, given a total ordering function over the set elements.

Spécification

- l'ordre sur les éléments

  module type `OrderedType` = sig .. end

- signature du module attendu

  module type `S` = sig .. end

- foncteur pour un type d'éléments donné

  functor `(Ord:OrderedType)` -> `S` with type `key` = `Ord.t`

Module Set

Utilisation

```
# module OrderedInt =
  struct type t=int let compare = compare end ;;
module OrderedInt : sig type t = int val compare : 'a -> 'a -> int end
# module IntSet = Set.Make(OrderedInt)
module IntSet :
  sig
    type elt = OrderedInt.t
    type t = Set.Make(OrderedInt).t
    val empty : t
    ...
```

Ou, plus court:

```
module IntSet = Set.Make(struct type t=int let compare = compare end)
```

Abstraction *vs* polymorphisme

Pourquoi `OrderedType` n'est-il pas simplement

```
module type OrderedType =
  sig val compare : 'a -> 'a -> int end
```

Supposons que cela soit et posons (pour faire simple)

```
module OrderedList (Elt:OrderedType) =
  struct
    let cons x xs = List.sort Elt.compare (x::xs)
  end
```

Application des listes ordonnées de dates

Un ordre sur les dates

Ordre lexicographique inversé

```
# module OrderedDate =
  struct
    let compare d1 d2 =
      let tuple_of d =
        (Date.get_year d, Date.get_month d, Date.get_day d)
      in
        compare (tuple_of d1) (tuple_of d2)
  end ;;

module OrderedDate : sig val compare : Date.t -> Date.t -> int end
```

Listes ordonnées de dates

Par application du foncteur

```
# module OrderedDates = OrderedList(OrderedDate) ;;

Signature mismatch:
Modules do not match:
  sig val compare : Date.t -> Date.t -> int end
is not included in
  OrderedType
Values do not match:
  val compare : Date.t -> Date.t -> int
is not included in
  val compare : 'a -> 'a -> int
```

Pourquoi ? avoir posé `OrderedType.compare` comme fonction polymorphe l'exige de ses instances.

Paramètre de type

- type abstrait ≈ paramètre de type

```
#module type OrderedType = sig
   type t val
   compare : t -> t -> int
  end ;;
[...]
# module OrderedDate = struct
    type t = Date.t
    let compare d1 d2 = [...]
  end
[...]
# module OrderedDates = OrderedList(OrderedDate) ;;
module OrderedDates :
  sig
    val cons : OrderedDate.t -> OrderedDate.t list -> OrderedDate.t list
  end
```

Deuxième partie

L'extension objets d'Objective Caml

Objets

Vocabulaire

- Classe: spécification d'un ensemble d'objets
- Objet: élément ou ***instance*** de classe
- Héritage: relation d'extension/spécialisation entre classes
- Attribut ou champs: données nommées d'une classe ou d'une instance
- Méthode: fonction appartenant à une classe ou une instance
- Appel (de méthode): activation (du code) d'une méthode

Objets en Objective Caml

Déclaration de classe

Syntaxe:

Class $id\ id_1 \ldots id_n =$
  object
  $\ldots$
    val $id = expr$
  $\ldots$
    val mutable $id = expr$
  $\ldots$
    method $id\ id_1 \ldots id_n = expr$
  $\ldots$
  end

Classes

Déclaration

La classe des compteurs

```
# class cpt =
  object
    val mutable c = 0
    method incr () = c <- c+1
    method reset () = c <- 0
    method get () = c
  end
;;
```

- les données: valeur entière modifiable `c`
- les traitements: méthodes `incr` `reset` `get`

Classes

$\approx$ modules

- Encapsulation données/traitements

$\Rightarrow$ fermeture

$\not\approx$ modules

- différenciation données/traitement

· variable d'instance (`val`) / méthode (`method`)

\+ possibilité de création de ***plusieurs instances***

· variables d'instance propre / méthodes partagées

Classes et types

Inférence statique

```
class cpt : object
    method get : unit -> int
    method incr : unit -> unit
    method reset : unit -> unit
    val mutable c : int
  end
```

≈ modules

- structure / signature
  - · déclaration de classe / nom et type des méthodes

Nota: bien qu'affichées, les variables (`val`) ne font pas partie du type.

$\Rightarrow$ masquage/abstraction des champs de données

Classes et objets

Création d'instance

- déclaration de classe

⇒ structure offrant les méthodes

+ ***constructeur*** d'instance

classes $\not\approx$ modules

Syntaxe: **new** *className*

```
# let c = new cpt ;;
let c = new cpt ;;
```

Nous y reviendrons.

Classes avec paramètres

Compteur avec valeur initiale et incrément

```
# class cpt c0 d =  object
    val mutable c = c0
    method incr () = c <- c+d
    method reset () = c <- c0
    method get () = c
  end ;;
class cpt : int -> int -> object
    method get : unit -> int
    method incr : unit -> unit
    method reset : unit -> unit
    val mutable c : int
  end
```

`int -> int -> object ... end`

Type fonctionnel = type du ***constructeur d'instance***

Classes et objets

Création d'instance (bis)

Syntaxe: **new** $id\ \ exp_1 \ldots exp_n$

- **new** mot clé
- id nom de la classe
- $exp_1 \ldots exp_n$ valeurs initiales pour les paramètres

```
# let c = new cpt 0 1 ;;
val c : cpt = <obj>
```

Classes et objets

Usage

Syntaxe: $exp\#id\ \ exp_1 \ldots exp_n$

- exp une instance et $\#$ symbole réservé
- id nom de la méthode et $exp_1 \ldots exp_n$ ses arguments

```
# c#get() ;;
- : int = 0
# c#incr() ; c#get() ;;
- : int = 1
```

Attention par d'accés aux variables

```
# c#c ;;
This expression has type cpt
It has no method c
```

Classe et classe

Relation d'héritage

Syntaxe: **inherit** $id \ exp_1 \ldots exp_n$

· **inherit** mot clef

· id nom de la ***classe mère***

· $exp_1 \ldots exp_n$ paramètre des création

$\approx$ module

- directive **include**

$\Rightarrow$ les variables et méthodes de la classe mère seront aussi celles de la ***classe fille***

Relation d'héritage

Extension des traitements

```
class cpt1 c0 s = object
    inherit cpt c0 s
    method to_string = Printf.sprintf "< init=%d; step=%d; value=%d >"
                           c0 s c
  end
```

Variables de la classe mère accessibles dans la définition.

```
# let c = new cpt1 0 1 ;;
val c : cpt1 = <obj>
# c#incr(); c#get() ;;
- : int = 1
# c#to_string ;;
- : string = "< init=0; step=1; value=1 >"
```

Méthodes de la classe mère+fille utilisables.

Relation à soit même

- Une mèthode ne peut être invoquée hors d'une instance/objet

· une classe veut définir un objet qui utilise ses propres méthodes

⇒ nommer localement les instances: auto référence

Syntaxe: **object (** *id* **)**

```
class gensym =
  object(self)
    inherit cpt 0 1
    val txt = "X"
    method sym = txt^(string_of_int c)
    method next = self#incr(); self#sym
  end
```

Choix du nom arbitraire, usage: `self`

Constructeur d'instance

Initialisation

Exécuter du code à la création

Syntaxe: **initializer** *exp*

```
# class verbose_gensym = object(self)
    inherit gensym
    initializer
      Printf.printf "Hello, I'm a new gensym for %s symbols\n" txt;
      Printf.printf "my initial value is %s\n" self#sym
  end
[...]
# let vs = new verbose_gensym ;;
Hello, I'm a new gensym for X symbols
my initial value is X0
val vs : verbose_gensym = <obj>
```

Relation d'héritage

Redéfinition de traitement

Variables ou méthodes

```
# class gensym' = object(self)
    inherit gensym
    val txt = "Y"
  end ;;
[...]
# let s = new gensym in s#sym ;;
- : string = "X0"
# let s' = new gensym' in s'#sym ;;
- : string = "Y0"
```

« The behaviour changed in ocaml 3.10 (previous behaviour was hiding.) »

Redéfinition

Liaison tardive

Le code des méthodes est ***résolu à l'exécution***

De `gensym` à `gensym'`

- on a redéfinit `txt`
- on n'a pas redéfinit `sym`

⇒ le code de `sym` choisit la valeur de `txt` selon l'instance qui l'invoque

Liaison dynamique ($\neq$ liaison statique)

Redéfinition

Préservation du type

Rappel: la méthode `get` de `cpt`

```
# let c = new cpt 0 1 in c#get ;;
- : unit -> int = <fun>
```

Tentative (vouée à l'échec)

```
class wrong_gensym =
  object(self)
    inherit cpt 0 1
    val txt = "X"
    method get () = txt^(string_of_int c)
  end  ;;
This expression has type string but is here used with type int
```

• Typage statique = pas de surcharge •

Relation mère/fille

Redéfinition

Définir le nouveau avec l'ancien: expliciter la référence

Syntaxe: **Inherit** ... **as** *Id*

```
class gensym1 =
  object(self)
    inherit cpt1 0 1 as super
    val txt = "X"
    method to_string =
      Printf.sprintf "[ txt=\"%s\" %s ]" txt super#to_string
  end
```

Liaison statique au code de la classe mère

Héritage multiple

Extension par agglomération

```
class cpt c0 d = object
  val mutable c = c0
  method incr () = c <- c+d
  method reset () = c <- c0
  method get () = c
end
```

```
class mksym = object
  val txt = "X"
  method sym_of_num n =
    Printf.sprintf"%s%d" txt n
end
```

```
class gensym2 =  object(self)
    inherit cpt 0 1
    inherit mksym
    method next =
      self#incr(); self#sym_of_num c
  end
```

Classe abstraite

Abstraction de traitement

Syntaxe: **class virtual** *id* ...

Syntaxe: **method virtual** *id* **:** *ty*

Une classe générique d'objets imprimables

```
class virtual printable =
  object(self)
    method print = print_string self#to_string
    method virtual to_string : string
  end
```

Pas d'instance de classe abstraite

```
# new printable ;;
One cannot create instances of the virtual class printable
```

De l'abstrait au concret

L'abstrait est héréditaire

```
class virtual printable_gensym =
  object
    inherit gensym
    inherit printable
  end
```

jusqu'à ce que l'on en décide autrement

```
class gensym4 =
  object
    inherit printable_gensym
    method to_string =
      Printf.sprintf"< text=%s value=%d>" txt c
  end
;;
```

Classe et type

Les classes comme type polymorphe

Syntaxe: **class [** 'id_1**,** $\ldots$**,** 'id_n **]** id $\ldots$

avec 'id_1, $\ldots$, 'id_n ***paramètres de types*** (variables)

```
class  ['a] stack = object
    val mutable s = ([] : 'a list)
    method push x = s <- x::s
    method pop =
      match s with
          [] -> failwith "Empty stack"
        | x::s' -> (s <- s'; x)
  end
```

$\Rightarrow$ ***expliciter*** par contrainte l'usage du paramètre de type

Héritage et classe paramétrée

Instanciation de type

Syntaxe: **inherit [** *ty* **]** *id*

```
# class  op_stack = object
    inherit [int] stack
    method add =
      match s with
          n1::n2::s' -> s <- (n1+n2)::s'
        | _ -> ()
  end ;;
class int_stack : object
    method add : unit
    method pop : int
    method push : int -> unit
    val mutable s : int list
  end
```

Héritage et polymorphisme

Rester générique

```
# class  ['a] f_stack =
  object
    inherit ['a] stack
    method app f =
      match s with
          x1::x2::s' -> s <- (f x1 x2)::s'
        | _ -> ()
  end ;;
class ['a] f_stack :
  object
    method app : ('a -> 'a -> 'a) -> unit
    method pop : 'a
    method push : 'a -> unit
    val mutable s : 'a list
  end
```

Classe et type

Les classes sont utilisables comme des types

```
class int_stack_stack =
  object
    inherit [int_stack] stack
  end
```

Les classes paramétrées sont des types paramétrés

```
class ['a] stack_stack =
  object
    inherit ['a stack] stack
  end
```

Notez: l'absence des `[ ]` dans ce cas.

Classes paramétrées et classes abstraites

Une classe abstraite peut instancier un paramètre de classe

```
class  printable_stack =
  object
    inherit [printable] stack
  end
```

La classe n'est pas abstraite: création d'instance

```
# let s = new printable_stack ;;
val s : printable_stack = <obj>
```

Mais il faudra des instances concrètes ***respectant*** `printable`

```
# s#push;;
- : printable -> unit = <fun>
```

Classe et type

Leur égalité

Le type défini par la classe est la liste (des noms) de ses méthodes avec leur type

```
# class printable_int n = object
   inherit printable
   method to_string = string_of_int n
  end ;;
class printable_int :
  int -> object method print : unit method to_string : string end
```

Du point de vue des types: `printable` = `printable_int`

```
# s#push (new printable_int 0) ;;
- : unit = ()
# s#pop#print ;;
0- : unit = ()
```

Classe et type

Leur in-égalité

Un entier affichable, et plus

```
# class int_obj n = object
    inherit printable
    method to_string = string_of_int n
    method to_float = float_of_int n
  end ;;
```

Les instances de `int_obj` ne sont pas de simples `printable`

```
# s#push (new int_obj 0) ;;
This expression has type int_obj but is here used with type printable
The second object type has no method to_float
```

QUE FAIRE ?

Leurs relations

Du point de vue des types:

- tout ce qui est `printable` doit posséder une méthode `to_string`
- or tout ce qui est `int_obj` possède une méthode `to_string`

⇒ donc tout ce qui est `int_obj` peut être vu comme `to_string`

• sous-typage •

Syntaxe: (*exp* :> *ctype*)

```
# s#push ((new int_obj 0) :> printable) ;;
- : unit = ()
```

Sous typage

Contrainte statique

Ce qui a été fait ne peut plus être défait

```
# s#pop#to_float;;
This expression has type printable
It has no method to_float
```

La sûreté a un prix

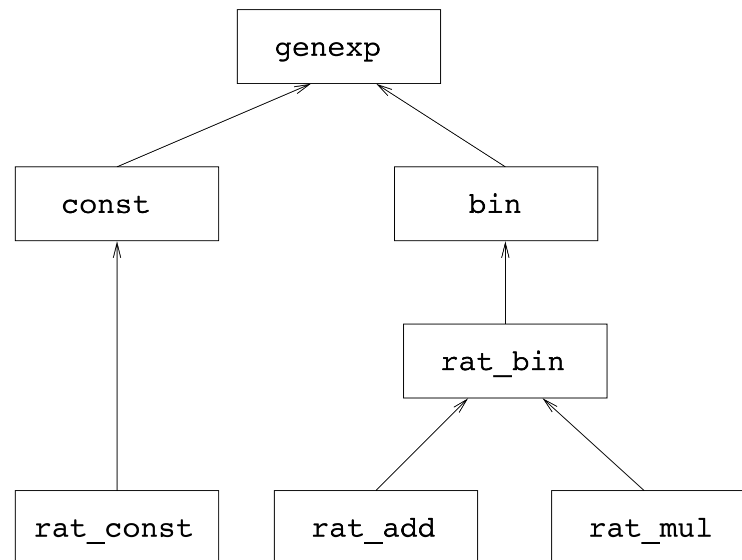
Rien ne doit être accessible hors de ce qui est prévu

Conception objet

Des expressions génériques aux expressions rationnelles

Expressions: entités qui fournissent

- une méthode d'évaluation
- une méthode d'affichage



Classe abstraite paramétrée

Motivation

- abstraite: classe générique pour les divers cas à venir (constante, opération binaire)
- paramétré: type générique des valeurs

Définition

```
class virtual ['value] genexp =
  object
    method virtual eval : 'value
    method virtual print : unit
  end
```

Les constantes abstraites

- une valeur est nécessaire pour construire une constante

⇒ méthode d'évaluation immédiate

Méthode d'évaluation déjà définissable

```
# class virtual ['value] const x =
  object
    inherit ['value] genexp
    method eval = x
  end ;;
class virtual ['a] const :
  'a -> object method eval : 'a method virtual print : unit end
```

Les opérations binaires abstraites

Méthodes ***privées***

- `op`: l'opération abstraite à deux valeurs associe une valeur
- `printop`: affichera l'opérateur
- `lpar` et `rpar`: méthodes de service

```
class virtual ['value] binexp e1 e2 =
  object(self)
    inherit ['value] genexp
    method virtual private op : 'value -> 'value -> 'value
    method eval = self#op e1#eval e2#eval
    method virtual private printop : unit
    method private lpar = print_string "("
    method private rpar = print_string ")"
    method print =
      self#lpar; e1#print; self#printop; e2#print; self#rpar
  end
```

Classes et inférence de type

Les arguments `e1` et `e2` doivent être instances d'une classe qui fournit les méthodes `eval` et `print`

```
class virtual ['a] binexp :
  < eval : 'a; print : 'b; .. > ->
  < eval : 'a; print : 'c; .. > ->
  object
    method eval : 'a
    [...]
  end
```

Notation pour les types-objets

```
< eval :  'a; print :  'b; ..  >
```

Nous y reviendrons

Arithmétique rationnelle

Les constantes

Détermination complète: paramètre et dernière méthode virtuelle

```
class rat_const n d =
  object
    inherit [int * int] const (n,d)
    initializer if d=0 then raise Division_by_zero
    method print = Printf.printf"(%d/%d)" n d
  end ;;
class rat_const :
  int -> int -> object method eval : int * int method print : unit end
```

Notez: l'utilisation de `initializer`

```
# let r1 = new rat_const 3 4 ;;
val r1 : rat_const = <obj>
# new rat_const 9 0;;
Exception: Division_by_zero.
```

Opérations binaires sur les rationnels

Redéfinition de `eval` pour réduction

```
class virtual rat_bin e1 e2 =
  object(self)
    inherit [int * int] binexp e1 e2 as super
    method private gcd x y =
      if y=0 then x else self#gcd y (x mod y)
    method eval =
      let (x,y) = super#eval in
      let d = self#gcd x y in
        (x/d, y/d)
  end
```

Nota: la méthode `eval` de `binexp (super)` n'est pas abstraite donc elle est invocable

Opérations binaires concrètes

L'addition

```
class rat_add e1 e2 =
  object
    inherit rat_bin e1 e2
    method printop = print_char '+'
    method op (x1,y1) (x2,y2) =
      (x1*y2 + x2*y1, y1*y2)
  end
```

La multiplication

```
class rat_mul e1 e2 =
  object
    inherit rat_bin e1 e2
    method printop = print_char '*'
    method op (x1,y1) (x2,y2) =
      (x1*x2, y1*y2)
  end
```

Utilisation

Une fonction eval/print pour les expressions d'arithmétique rationnelle

```
let eval_print_rat e =
  e#print;
  let (x,y) = e#eval in
    Printf.printf" = (%d,%d)\n" x y
```

Quelques utilitaires et leur application

```
# let rcst x y = new rat_const x y in
  let rplus x y = new rat_add x y in
  let rmult x y = new rat_mul x y in
  let e = (rmult (rplus (rcst 1 3) (rcst 3 5)) (rcst 1 2)) in
    eval_print_rat e ;;
(((1/3)+(3/5))*(1/2)) = (7,15)
- : unit = ()
```

Classes, objets et types

Du type des expressions

```
# let r_1_2 = new rat_const 1 2 ;;
val r_1_2 : rat_const = <obj>
# let e1 = new rat_add r_1_2 r_1_2 ;;
val e1 : rat_add = <obj>
# let e2 = new rat_mul e1 r_1_2 ;;
val e2 : rat_mul = <obj>
```

Le type des instances est déterminé par le constructeur utilisé

classe $\Rightarrow$ type

A priori non compatibles

```
# [r_1_2; e1; e2];;
This expression has type rat_add but is here used with type rat_const
The second object type has no method op
```

Classes, objets et type

Inférence du type le plus général

Pourtant `rat_mul` peut recevoir aussi bien des instances de `rat_const` que `rat_add`, etc.

Pourquoi ? Parce que:

```
# let rmult = new rat_mul ;;
val rmult :
  < eval : int * int; print : 'a; .. > ->
  < eval : int * int; print : 'b; .. > -> rat_mul = <fun>
```

type ouvert

avec : .. ≈ variable de type

Type ouvert

Compatibilité

```
# class c0 =
  object(self) method txt = "Hello" method m = print_string self#txt end
;;
class c0 :
   object method m : unit method txt : string end
# class c1 o =
  object method m = o#m; print_string " world" end
;;
class c1 :
  < m : 'a; .. > -> object method m : unit end
# let o = new c1 (new c0) ;;
val o : c1 = <obj>
# o#m ;;
Hello world- : unit = ()
```

Typage

de `new c1 (new c0)`

- `new c1` attend un `< m : 'a; .. >`
- `new c0` fournit `< m : unit; txt : string >`
- `< m : 'a` prend `< m : unit` car `'a` prend `unit`
- et `.. >` prend le reste `txt : string >`

Note compatibilité $\neq$ héritage

Compatibilité et sous typage

Rendre compatible

```
[r_1_2; (e1:>rat_const); (e2:>rat_const)] ;;
- : rat_const list = [<obj>; <obj>; <obj>]
```

Mieux: un ***type générique*** pour les rationnels

```
type rat_genexp = (int*int) genexp ;;
```

Plus petit ensemble des méthodes exigibles

Homogénéïsation des diverses instances

```
[(r_1_2:>rat_genexp); (e1:>rat_genexp); (e2:>rat_genexp)] ;;
- : rat_genexp list = [<obj>; <obj>; <obj>]
```

Interface

Spécification de type objet

Syntaxe:

class type Id = **object val** Id_1: t_1 **method** Id_2: t_2

```
class type rat_genexp =
  object
    method print : unit
    method eval : int*int
  end
```

Utilisables partout où les types sont utilisables.

Le type de `self`

Définition récursive

Référence au type de l'objet lui-même, encore indéterminé

$\Rightarrow$ variable de type

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

La méthode `eq` prend en argument une instance de sa classe

Héritage et sous typage

Leur absence de relation

Être sous classe sans être sous type

```
# class c1 (x:int) =
  object
    inherit equalizable
    method get_x = x
    method eq o = (o#get_x = x)
  end ;;
class c1 : int ->
   object ('a) method eq : 'a -> bool method get_x : int end
```

Ici aussi, la méthode `eq` prend en argument une instance de sa classe: `c1`

Héritage et sous typage (suite)

On ne peut restreindre les instances de `c1` au type `equalizable`

```
# ((new c1 0) :> equalizable) ;;
This expression cannot be coerced to type
  equalizable = < eq : equalizable -> bool >;
it has type c1 = < eq : c1 -> bool; get_x : int > but is here used with ty
  < eq : equalizable -> bool; .. >
Type c1 = < eq : c1 -> bool; get_x : int > is not compatible with type
  equalizable = < eq : equalizable -> bool >
The second object type has no method get_x
```

Le type de `eq` dans `c1` n'est pas sous type du type de `eq` dans `equalizable`

car `equalizable` n'est pas sous type de `c1` (`no method get_x`)