

OCaml: objets avancés

David Delahaye

Faculté des Sciences
David.Delahaye@lirmm.fr

Licence L3 2018-2019

Plan du cours

3 semaines de cours

- ❶ Noyau fonctionnel ;
- ❷ Objets simples (héritage simple et multiple, sous-typage) ;
- ❸ *Objets avancés (types ouverts, contraintes, « self-types »).*

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class virtual int_expr =  
  object  
    method virtual eval : int  
  end;;  
class virtual int_expr :  
  object method virtual eval : int end  
# class cte n =  
  object  
    inherit int_expr  
    val content = n  
    method eval = content  
  end;;  
class cte :  
  int →  
  object val content : int method eval : int end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class virtual int_expr =  
  object  
    method virtual eval : int  
  end;;  
class virtual int_expr :  
  object method virtual eval : int end  
# class cte n =  
  object  
    inherit int_expr  
    val content = n  
    method eval = content  
  end;;  
class cte :  
  int →  
  object val content : int method eval : int end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class virtual int_expr =  
  object  
    method virtual eval : int  
  end;;  
class virtual int_expr :  
  object method virtual eval : int end  
# class cte n =  
  object  
    inherit int_expr  
    val content = n  
    method eval = content  
  end;;  
class cte :  
  int →  
  object val content : int method eval : int end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class virtual int_expr =  
  object  
    method virtual eval : int  
  end;;  
class virtual int_expr :  
  object method virtual eval : int end  
# class cte n =  
  object  
    inherit int_expr  
    val content = n  
    method eval = content  
  end;;  
class cte :  
  int →  
  object val content : int method eval : int end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class virtual bin_op =  
object  
  inherit int_expr  
  val virtual left : int_expr  
  val virtual right : int_expr  
end;;  
  
class virtual bin_op :  
  object  
    val virtual left : int_expr  
    val virtual right : int_expr  
    method virtual eval : int  
  end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class virtual bin_op =  
object  
  inherit int_expr  
  val virtual left : int_expr  
  val virtual right : int_expr  
end;;  
class virtual bin_op :  
  object  
    val virtual left : int_expr  
    val virtual right : int_expr  
    method virtual eval : int  
  end
```


Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class add l r =  
object  
  inherit bin_op  
  val left = l  
  val right = r  
  method eval = left#eval + right#eval  
end;;  
  
class add :  
  int_expr →  
  int_expr →  
  object val left : int_expr val right : int_expr  
    method eval : int end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# class add l r =  
  object  
    inherit bin_op  
    val left = l  
    val right = r  
    method eval = left#eval + right#eval  
  end;;  
class add :  
  int_expr →  
  int_expr →  
  object val left : int_expr val right : int_expr  
    method eval : int end
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# let e = new add (new cte 1) (new cte 2);;  
  val e : add = <obj>  
# e#eval;;  
- : int = 3
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# let e = new add (new cte 1) (new cte 2);;  
val e : add = <obj>  
# e#eval;;  
- : int = 3
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# let e = new add (new cte 1) (new cte 2);;  
val e : add = <obj>  
# e#eval;;  
- : int = 3
```

Variables d'instance, méthodes et classes virtuelles

Expressions de la logique propositionnelle

```
# let e = new add (new cte 1) (new cte 2);;  
val e : add = <obj>  
# e#eval;;  
- : int = 3
```

Principe

- Apparaissent dans le type de la classe mais pas dans les instances ;
- Ne peuvent être appelées que dans d'autres méthodes de la classe ;
- Ne peuvent pas être envoyées à des instances de la classe ;
- Sont héritées et donc utilisables dans les méthodes des sous-classes.

Méthodes privées

Comptes bancaires

```
# class great_account b =  
object (self)  
  val mutable balance = b  
  method get = balance  
  method private bonus n = balance <- balance +. n  
  method deposit a =  
    balance <- balance +. a; self#bonus 5.  
  method withdraw a = balance <- balance -. a  
end;;
```


Méthodes privées

Comptes bancaires

```
class great_account :  
  float →  
  object  
    val mutable balance : float  
    method private bonus : float → unit  
    method deposit : float → unit  
    method get : float  
    method withdraw : float → unit  
  end
```

Méthodes privées

Comptes bancaires

```
# let a = new great_account 100.;;  
val a : great_account = <obj>  
# a#deposit 50.;;  
- : unit = ()  
# a#get;;  
- : float = 155.
```

Méthodes privées

Comptes bancaires

```
# let a = new great_account 100.;;  
val a : great_account = <obj>  
# a#deposit 50.;;  
- : unit = ()  
# a#get;;  
- : float = 155.
```

Méthodes privées

Comptes bancaires

```
# let a = new great_account 100.;;  
val a : great_account = <obj>  
# a#deposit 50.;;  
- : unit = ()  
# a#get;;  
- : float = 155.
```

Méthodes privées

Comptes bancaires

```
# let a = new great_account 100.;;  
val a : great_account = <obj>  
# a#deposit 50.;;  
- : unit = ()  
# a#get;;  
- : float = 155.
```

Méthodes privées

Comptes bancaires

```
# let a = new great_account 100.;;  
val a : great_account = <obj>  
# a#deposit 50.;;  
- : unit = ()  
# a#get;;  
- : float = 155.
```

Méthodes privées

Comptes bancaires

```
# let a = new great_account 100.;;  
val a : great_account = <obj>  
# a#deposit 50.;;  
- : unit = ()  
# a#get;;  
- : float = 155.
```

Méthodes privées

Comptes bancaires

```
# class not_so_great_account b =  
object (self)  
  inherit great_account b  
  method withdraw a =  
    balance <- balance -. a; self#bonus (-5.)  
end;;
```


Méthodes privées

Comptes bancaires

```
class not_so_great_account :  
  float →  
  object  
    val mutable balance : float  
    method private bonus : float → unit  
    method deposit : float → unit  
    method get : float  
    method withdraw : float → unit  
  end
```

Méthodes privées

Comptes bancaires

```
# let b = new not_so_great_account 100.;;  
val b : not_so_great_account = <obj>  
# b#withdraw 50.;;  
- : unit = ()  
# b#get;;  
- : float = 45.
```

Méthodes privées

Comptes bancaires

```
# let b = new not_so_great_account 100.;;  
val b : not_so_great_account = <obj>  
# b#withdraw 50.;;  
- : unit = ()  
# b#get;;  
- : float = 45.
```

Méthodes privées

Comptes bancaires

```
# let b = new not_so_great_account 100.;;  
val b : not_so_great_account = <obj>  
# b#withdraw 50.;;  
- : unit = ()  
# b#get;;  
- : float = 45.
```

Méthodes privées

Comptes bancaires

```
# let b = new not_so_great_account 100.;;  
val b : not_so_great_account = <obj>  
# b#withdraw 50.;;  
- : unit = ()  
# b#get;;  
- : float = 45.
```

Méthodes privées

Comptes bancaires

```
# let b = new not_so_great_account 100.;;  
val b : not_so_great_account = <obj>  
# b#withdraw 50.;;  
- : unit = ()  
# b#get;;  
- : float = 45.
```

Méthodes privées

Comptes bancaires

```
# let b = new not_so_great_account 100.;;  
val b : not_so_great_account = <obj>  
# b#withdraw 50.;;  
- : unit = ()  
# b#get;;  
- : float = 45.
```

Types ouverts

Dans les fonctions

```
# let add o = o#get +. 10.;;  
val add : < get : float; .. > → float = <fun>
```

- La fonction `add` attend un objet qui contient au moins la méthode `get` ;
- C'est une autre forme de polymorphisme paramétrique, car la fonction peut recevoir des objets d'une infinité de types.

Types ouverts

Dans les fonctions

```
# let add o = o#get +. 10.;;  
val add : < get : float; .. > → float = <fun>
```

- La fonction `add` attend un objet qui contient au moins la méthode `get` ;
- C'est une autre forme de polymorphisme paramétrique, car la fonction peut recevoir des objets d'une infinité de types.

Types ouverts

Dans les fonctions

```
# class account b =  
object  
  val mutable balance = b  
  method get = balance  
  method deposit a = balance <- balance +. a  
  method withdraw a = balance <- balance -. a  
end;;  
  
class account :  
  float →  
  object  
    val mutable balance : float  
    method deposit : float → unit  
    method get : float  
    method withdraw : float → unit  
  end
```

Types ouverts

Dans les fonctions

```
# class account b =  
object  
  val mutable balance = b  
  method get = balance  
  method deposit a = balance <- balance +. a  
  method withdraw a = balance <- balance -. a  
end;;  
class account :  
  float →  
  object  
    val mutable balance : float  
    method deposit : float → unit  
    method get : float  
    method withdraw : float → unit  
  end
```

Types ouverts

Dans les fonctions

```
# class ['a] cell (n : 'a) =  
object  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
end;;  
  
class ['a] cell :  
  'a →  
  object val mutable content : 'a  
    method get : 'a method set : 'a → unit end
```

Types ouverts

Dans les fonctions

```
# class ['a] cell (n : 'a) =  
object  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
end;;  
class ['a] cell :  
  'a →  
  object val mutable content : 'a  
    method get : 'a method set : 'a → unit end
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```


Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les fonctions

```
# let a = new account 100.;;  
val a : account = <obj>  
# let c = new cell 50.;;  
val c : float cell = <obj>  
# add a;;  
- : float = 110.  
# add c;;  
- : float = 60.
```

Types ouverts

Dans les classes

```
# class ['a] cell (n : 'a) =  
object  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
  method print = content#print  
end;;  
  
class ['a] cell :  
  'a →  
  object  
    constraint 'a = < print : 'b; .. >  
    val mutable content : 'a  
    method get : 'a  
    method print : 'b  
    method set : 'a → unit  
  end
```

Types ouverts

Dans les classes

```
# class ['a] cell (n : 'a) =  
object  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
  method print = content#print  
end;;  
class ['a] cell :  
  'a →  
  object  
    constraint 'a = < print : 'b; .. >  
    val mutable content : 'a  
    method get : 'a  
    method print : 'b  
    method set : 'a → unit  
  end
```

Types ouverts

Dans les classes

La contrainte `constraint 'a = < print : 'b; .. >` indique que :

- La classe attend en paramètre un objet qui contient au moins la méthode `print` de type `'b`;
- Problème : l'inférence de type n'a pas assez d'information pour contraindre correctement le type de `print`, qui est trop général;
- Pour contraindre mieux le type de `print` (on aimerait `unit`) :
 - Soit on indique la contrainte explicitement dans la classe;
 - Soit on utilise une autre classe pour contraindre le type.

Types ouverts

Dans les classes

La contrainte `constraint 'a = < print : 'b; .. >` indique que :

- La classe attend en paramètre un objet qui contient au moins la méthode `print` de type `'b`;
- Problème : l'inférence de type n'a pas assez d'information pour contraindre correctement le type de `print`, qui est trop général;
- Pour contraindre mieux le type de `print` (on aimerait `unit`) :
 - Soit on indique la contrainte explicitement dans la classe;
 - Soit on utilise une autre classe pour contraindre le type.

Types ouverts

Dans les classes

La contrainte `constraint 'a = < print : 'b; .. >` indique que :

- La classe attend en paramètre un objet qui contient au moins la méthode `print` de type `'b`;
- Problème : l'inférence de type n'a pas assez d'information pour contraindre correctement le type de `print`, qui est trop général;
- Pour contraindre mieux le type de `print` (on aimerait `unit`) :
 - Soit on indique la contrainte explicitement dans la classe;
 - Soit on utilise une autre classe pour contraindre le type.

Types ouverts

Dans les classes

La contrainte `constraint 'a = < print : 'b; .. >` indique que :

- La classe attend en paramètre un objet qui contient au moins la méthode `print` de type `'b`;
- Problème : l'inférence de type n'a pas assez d'information pour contraindre correctement le type de `print`, qui est trop général;
- Pour contraindre mieux le type de `print` (on aimerait `unit`) :
 - ▶ Soit on indique la contrainte explicitement dans la classe;
 - ▶ Soit on utilise une autre classe pour contraindre le type.

Types ouverts

Dans les classes

La contrainte `constraint 'a = < print : 'b; .. >` indique que :

- La classe attend en paramètre un objet qui contient au moins la méthode `print` de type `'b`;
- Problème : l'inférence de type n'a pas assez d'information pour contraindre correctement le type de `print`, qui est trop général;
- Pour contraindre mieux le type de `print` (on aimerait `unit`) :
 - ▶ Soit on indique la contrainte explicitement dans la classe;
 - ▶ Soit on utilise une autre classe pour contraindre le type.

Types ouverts

Contrainte explicite dans la classe

```
# class ['a] cell (n : 'a) =  
object  
  constraint 'a = < print : unit; .. >  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
  method print = content#print  
end;;
```

Types ouverts

Contrainte explicite dans la classe

```
class ['a] cell :  
  'a →  
  object  
    constraint 'a = < print : unit; .. >  
    val mutable content : 'a  
    method get : 'a  
    method print : unit  
    method set : 'a → unit  
  end
```

Types ouverts

Contrainte explicite dans la classe

```
# class account b =  
object  
  val mutable balance = b  
  method get = balance  
  method deposit a = balance <- balance +. a  
  method withdraw a = balance <- balance -. a  
  method print = print_float balance; print_newline()  
end;;
```

Types ouverts

Contrainte explicite dans la classe

```
class account :  
  float →  
  object  
    val mutable balance : float  
    method deposit : float → unit  
    method get : float  
    method print : unit  
    method withdraw : float → unit  
  end
```


Types ouverts

Contrainte explicite dans la classe

```
# class point (xi , yi) =  
object  
  val x = xi  
  val y = yi  
  method get = (x, y)  
  method print =  
    print_string "(";  
    print_int x;  
    print_string ", ";  
    print_int y;  
    print_endline ")"  
end;;
```

Types ouverts

Contrainte explicite dans la classe

```
class point :  
  int * int →  
  object  
    val x : int  
    val y : int  
    method get : int * int  
    method print : unit  
  end
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
  val a : account = <obj>  
# let p = new point (1, 2);;  
  val p : point = <obj>  
# let ca = new cell a;;  
  val ca : account cell = <obj>  
# let cp = new cell p;;  
  val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```


Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte explicite dans la classe

```
# let a = new account 100.;;  
val a : account = <obj>  
# let p = new point (1, 2);;  
val p : point = <obj>  
# let ca = new cell a;;  
val ca : account cell = <obj>  
# let cp = new cell p;;  
val cp : point cell = <obj>  
# ca#print;;  
100.  
- : unit = ()  
# cp#print;;  
(1, 2)  
- : unit = ()
```

Types ouverts

Contrainte avec une autre classe

```
# class virtual printable =  
object  
  method virtual print : unit  
end;;  
  
class virtual printable :  
  object method virtual print : unit end
```

Types ouverts

Contrainte avec une autre classe

```
# class virtual printable =  
object  
  method virtual print : unit  
end;;  
class virtual printable :  
  object method virtual print : unit end
```


Types ouverts

Contrainte avec une autre classe

```
# class  ['a] cell (n : 'a) =  
object  
  constraint 'a = #printable  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
  method print = content#print  
end;;
```

Types ouverts

Contrainte avec une autre classe

```
class ['a] cell :  
  'a →  
  object  
    constraint 'a = #printable  
    val mutable content : 'a  
    method get : 'a  
    method print : unit  
    method set : 'a → unit  
  end
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# class ['a] cell (n : 'a) =  
object  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
end;;  
  
class ['a] cell :  
  'a →  
  object val mutable content : 'a  
    method get : 'a method set : 'a → unit end
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# class ['a] cell (n : 'a) =  
object  
  val mutable content = n  
  method get = content  
  method set n = content <- n  
end;;  
class ['a] cell :  
  'a →  
  object val mutable content : 'a  
    method get : 'a method set : 'a → unit end
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# class int_cell n =  
  object  
    inherit [int] cell n  
  end;;  
  
class int_cell :  
  int →  
  object  
    val mutable content : int  
    method get : int  
    method set : int → unit  
  end
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# class int_cell n =  
  object  
    inherit [int] cell n  
  end;;  
class int_cell :  
  int →  
  object  
    val mutable content : int  
    method get : int  
    method set : int → unit  
  end
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# let c = new int_cell 1;;  
val c : int_cell = <obj>  
# c#get;;  
- : int = 1
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# let c = new int_cell 1;;  
val c : int_cell = <obj>  
# c#get;;  
- : int = 1
```


Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# let c = new int_cell 1;;  
val c : int_cell = <obj>  
# c#get;;  
- : int = 1
```

Héritage et polymorphisme paramétrique

Spécialisation de la classe polymorphe

```
# let c = new int_cell 1;;  
val c : int_cell = <obj>  
# c#get;;  
- : int = 1
```

Héritage et polymorphisme paramétrique

Ajout d'une contrainte

```
# class ['a] printable_cell (n : 'a) =  
object  
  inherit [#printable] cell n  
  method print = content#print  
end;;  
  
class ['a] printable_cell :  
  'a →  
  object  
    constraint 'a = #printable  
    val mutable content : 'a  
    method get : 'a  
    method print : unit  
    method set : 'a → unit  
  end
```

Héritage et polymorphisme paramétrique

Ajout d'une contrainte

```
# class ['a] printable_cell (n : 'a) =  
object  
  inherit [#printable] cell n  
  method print = content#print  
end;;  
class ['a] printable_cell :  
  'a →  
object  
  constraint 'a = #printable  
  val mutable content : 'a  
  method get : 'a  
  method print : unit  
  method set : 'a → unit  
end
```

Héritage et polymorphisme paramétrique

Renforcement des contraintes

```
# class ['a] getable_printable_cell (n : 'a) =  
object  
  inherit ['a] printable_cell n  
  constraint 'a = < get : 'b; .. >  
  method print = content#print  
end;;  
  
class ['a] getable_printable_cell :  
  'a →  
object  
  constraint 'a = < get : 'b; print : unit; .. >  
  val mutable content : 'a  
  method get : 'a  
  method print : unit  
  method set : 'a → unit  
end
```

Héritage et polymorphisme paramétrique

Renforcement des contraintes

```
# class ['a] getable_printable_cell (n : 'a) =  
object  
  inherit ['a] printable_cell n  
  constraint 'a = < get : 'b; .. >  
  method print = content#print  
end;;  
class ['a] getable_printable_cell :  
  'a →  
object  
  constraint 'a = < get : 'b; print : unit; .. >  
  val mutable content : 'a  
  method get : 'a  
  method print : unit  
  method set : 'a → unit  
end
```

Bien typer les méthodes binaires

Égalité : en Java

```
class Object {  
    public boolean equals (Object obj) { ... }  
    ...  
}
```

Bien typer les méthodes binaires

Égalité : en Java

```
class Date {  
  
    int d, m, y;  
  
    public boolean equals (Object obj) {  
        Date a = (Date)obj;  
        return (d == a.d) && (m == a.m) && (y == a.y);  
    }  
  
    ...  
  
}
```

- Non satisfaisant car utilisation d'un « downcast » ;
- Erreurs de type à l'exécution possibles si le paramètre *obj* n'est pas un objet de type *Date*.

Bien typer les méthodes binaires

Égalité : en Java

```
class Date {  
  
    int d, m, y;  
  
    public boolean equals (Object obj) {  
        Date a = (Date)obj;  
        return (d == a.d) && (m == a.m) && (y == a.y);  
    }  
  
    ...  
}
```

- Non satisfaisant car utilisation d'un « downcast » ;
- Erreurs de type à l'exécution possibles si le paramètre *obj* n'est pas un objet de type *Date*.

Bien typer les méthodes binaires

Égalité : en Java

```
class Date {  
  
    int d, m, y;  
  
    public boolean equals (Object obj) {  
        Date a = (Date)obj;  
        return (d == a.d) && (m == a.m) && (y == a.y);  
    }  
  
    ...  
}
```

- Non satisfaisant car utilisation d'un « downcast » ;
- Erreurs de type à l'exécution possibles si le paramètre *obj* n'est pas un objet de type *Date*.

Bien typer les méthodes binaires

Égalité : en Java

```
class Date {  
  
    int d, m, y;  
  
    public boolean equals (Object obj) {  
        Date a = (Date)obj;  
        return (d == a.d) && (m == a.m) && (y == a.y);  
    }  
  
    ...  
  
}
```

- Non satisfaisant car utilisation d'un « downcast » ;
- Erreurs de type à l'exécution possibles si le paramètre *obj* n'est pas un objet de type *Date*.

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# class point (xi, yi) =  
  object (self : 'a)  
    val x = xi  
    val y = yi  
    method get = (x, y)  
    method equals (p : 'a) = (x, y) = p#get  
    method print =  
      print_string "(";  
      print_int x;  
      print_string ", "  
      print_int y;  
      print_endline ")"  
  end;;
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
class point :  
  int * int →  
  object ('a)  
    val x : int  
    val y : int  
    method equals : 'a → bool  
    method get : int * int  
    method print : unit  
  end
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# class colored_point (xi, yi) (c : string) =  
object (self : 'a)  
  inherit point (xi, yi)  
  val color = c  
  method get_color = color  
  method equals (p : 'a) =  
    (x, y) == p#get && c == p#get_color  
end;;
```

Solution d'OCaml : les « self-types » (types récur­sifs)

Égalité : en OCaml

```
class colored_point :  
  int * int →  
  string →  
  object ('a)  
    val color : string  
    val x : int  
    val y : int  
    method equals : 'a → bool  
    method get : int * int  
    method get_color : string  
    method print : unit  
  end
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```


Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false

# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false

# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false

# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récur­sifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false

# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```


Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false

# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récur­sifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# let p1 = new point (1, 2);;
val p1 : point = <obj>
# let p2 = new point (2, 1);;
val p2 : point = <obj>
# p1#equals p2;;
- : bool = false
# let cp1 = new colored_point (1, 2) "blue";;
val cp1 : colored_point = <obj>
# let cp2 = new colored_point (2, 1) "red";;
val cp2 : colored_point = <obj>
# cp1#equals cp2;;
- : bool = false
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# p1#equals cp1;;
```

*Error: This expression has **type** colored_point
but an expression was expected **of type** point
The second **object type** has no **method** get_color*

```
# cp1#equals p1;;
```

*Error: This expression has **type** point but an expression
was expected **of type** colored_point
The first **object type** has no **method** get_color*

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# p1#equals cp1;;
```

```
Error: This expression has type colored_point  
      but an expression was expected of type point  
      The second object type has no method get_color
```

```
# cp1#equals p1;;
```

```
Error: This expression has type point but an expression  
      was expected of type colored_point  
      The first object type has no method get_color
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# p1#equals cp1;;
```

*Error: This expression has **type** colored_point
but an expression was expected of **type** point
The second **object type** has no **method** get_color*

```
# cp1#equals p1;;
```

*Error: This expression has **type** point but an expression
was expected of **type** colored_point
The first **object type** has no **method** get_color*

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

```
# p1#equals cp1;;
```

```
Error: This expression has type colored_point  
      but an expression was expected of type point  
      The second object type has no method get_color
```

```
# cp1#equals p1;;
```

```
Error: This expression has type point but an expression  
      was expected of type colored_point  
      The first object type has no method get_color
```

Solution d'OCaml : les « self-types » (types récurifs)

Égalité : en OCaml

- Dans `(self : 'a)`, `'a` représente le type de la classe que l'on définit ;
- Cette déclaration doit être reportée dans les sous-classes pour typer correctement les méthodes qui l'utilisent (ici `equals`) ;
- De fait, la méthode `equals` de la classe `colored_point` est bien une redéfinition de la méthode `equals` de la classe `point`.

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# class ['a] other_point (xi , yi) =  
object  
  val x = xi  
  val y = yi  
  method get = (x, y)  
  method equals (p : 'a) = (x, y) = p#get  
  method print =  
    print_string "(";  
    print_int x;  
    print_string ", "  
    print_int y;  
    print_endline ")"  
end;;
```

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
class ['a] other_point :  
  int * int →  
  object  
    constraint 'a = < get : int * int; .. >  
    val x : int  
    val y : int  
    method equals : 'a → bool  
    method get : int * int  
    method print : unit  
  end
```

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# class couple (xi , yi) =  
object  
  val x = xi  
  val y = yi  
  method get = (x, y)  
  method print =  
    print_string "(";  
    print_int x;  
    print_string ", ";  
    print_int y;  
    print_endline ")"  
end;;
```

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
class couple :  
  int * int →  
  object  
    val x : int  
    val y : int  
    method get : int * int  
    method print : unit  
  end
```

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# let op = new other_point (1, 2);;  
val op : < get : int * int; _.. > other_point = <obj>  
# let c = new couple (1, 2);;  
val c : couple = <obj>  
# op#equals c;;  
- : bool = true
```

- On compare deux objets de types très différents !
- La fonction equals ne fait que vérifier que le paramètre c possède la méthode get.

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# let op = new other_point (1, 2);;  
val op : < get : int * int; _.. > other_point = <obj>  
# let c = new couple (1, 2);;  
val c : couple = <obj>  
# op#equals c;;  
- : bool = true
```

- On compare deux objets de types très différents !
- La fonction `equals` ne fait que vérifier que le paramètre `c` possède la méthode `get`.

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# let op = new other_point (1, 2);;  
val op : < get : int * int; _.. > other_point = <obj>  
# let c = new couple (1, 2);;  
val c : couple = <obj>  
# op#equals c;;  
- : bool = true
```

- On compare deux objets de types très différents !
- La fonction `equals` ne fait que vérifier que le paramètre `c` possède la méthode `get`.

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# let op = new other_point (1, 2);;  
val op : < get : int * int; _.. > other_point = <obj>  
# let c = new couple (1, 2);;  
val c : couple = <obj>  
# op#equals c;;  
- : bool = true
```

- On compare deux objets de types très différents !
- La fonction `equals` ne fait que vérifier que le paramètre `c` possède la méthode `get`.

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# let op = new other_point (1, 2);;  
val op : < get : int * int; _.. > other_point = <obj>  
# let c = new couple (1, 2);;  
val c : couple = <obj>  
# op#equals c;;  
- : bool = true
```

- On compare deux objets de types très différents !
- La fonction `equals` ne fait que vérifier que le paramètre `c` possède la méthode `get`.

Attention aux fausses méthodes binaires !

Deux classes qui se ressemblent beaucoup

```
# let op = new other_point (1, 2);;  
val op : < get : int * int; _.. > other_point = <obj>  
# let c = new couple (1, 2);;  
val c : couple = <obj>  
# op#equals c;;  
- : bool = true
```

- On compare deux objets de types très différents !
- La fonction `equals` ne fait que vérifier que le paramètre `c` possède la méthode `get`.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# class point (xi, yi) =  
  object (self : 'a)  
    val x = xi  
    val y = yi  
    method get = (x, y)  
    method equals (p : 'a) = (x, y) = p##get  
    method print =  
      print_string "(";  
      print_int x;  
      print_string ", "  
      print_int y;  
      print_endline ")"  
  end;;
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
class point :  
  int * int →  
  object ('a)  
    val x : int  
    val y : int  
    method equals : 'a → bool  
    method get : int * int  
    method print : unit  
end
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# class colored_point (xi , yi) (c : string) =  
  object (self : 'a)  
    inherit point (xi , yi)  
    val color = c  
    method get_color = color  
    method equals (p : 'a) =  
      (x, y) == p#get && c == p#get_color  
  end;;
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
class colored_point :  
  int * int →  
  string →  
  object ('a)  
    val color : string  
    val x : int  
    val y : int  
    method equals : 'a → bool  
    method get : int * int  
    method get_color : string  
    method print : unit  
end
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# let p = new point (1, 2);;
val p : point = <obj>
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# let l = [p; (cp :> point)];;
Error: Type
    colored_point =
      < equals : colored_point → bool;
        get : int * int;
        get_color : string; print : unit >
is not a subtype of
    point =
      < equals : point → bool;
        get : int * int;
        print : unit >
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# let p = new point (1, 2);;
val p : point = <obj>
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# let l = [p; (cp :> point)];;
Error: Type
    colored_point =
      < equals : colored_point → bool;
        get : int * int;
        get_color : string; print : unit >
is not a subtype of
    point =
      < equals : point → bool;
        get : int * int;
        print : unit >
```


Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# let p = new point (1, 2);;
val p : point = <obj>
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# let l = [p; (cp :> point)];;
Error: Type
    colored_point =
      < equals : colored_point → bool;
        get : int * int;
        get_color : string; print : unit >
is not a subtype of
    point =
      < equals : point → bool;
        get : int * int;
        print : unit >
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# let p = new point (1, 2);;
val p : point = <obj>
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# let l = [p; (cp :> point)];;
Error: Type
    colored_point =
      < equals : colored_point → bool;
        get : int * int;
        get_color : string; print : unit >
is not a subtype of
    point =
      < equals : point → bool;
        get : int * int;
        print : unit >
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# let p = new point (1, 2);;  
val p : point = <obj>  
# let cp = new colored_point (1, 2) "blue";;  
val cp : colored_point = <obj>  
# let l = [p; (cp :> point)];;
```

Error: Type

```
    colored_point =  
      < equals : colored_point → bool;  
        get : int * int;  
        get_color : string; print : unit >  
is not a subtype of  
    point =  
      < equals : point → bool;  
        get : int * int;  
        print : unit >
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

```
# let p = new point (1, 2);;
val p : point = <obj>
# let cp = new colored_point (1, 2) "blue";;
val cp : colored_point = <obj>
# let l = [p; (cp :> point)];;
Error: Type
    colored_point =
      < equals : colored_point → bool;
        get : int * int;
        get_color : string; print : unit >
is not a subtype of
    point =
      < equals : point → bool;
        get : int * int;
        print : unit >
```

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - Les méthodes de `point` sont bien dans `colored_point` ;
 - Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ✧ C'est le cas de `get` et `print` ;
 - ✧ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ✧ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- `colored_point` n'est pas un sous-type de `point` !
- Pourquoi ?
 - ▶ Les méthodes de `point` sont bien dans `colored_point` ;
 - ▶ Mais avec les mêmes types ?
 - ★ C'est le cas de `get` et `print` ;
 - ★ Pour `equals`, on a :
 $\text{equals}_{\text{point}} : 'a \rightarrow \text{bool}$, où $'a = \text{point}$
 $\text{equals}_{\text{colored_point}} : 'a \rightarrow \text{bool}$, où $'a = \text{colored_point}$
 - ★ Le type de `equalscolored_point` est un sous-type de celui de `equalspoint` si et seulement si :
Le $'a$ de `equalspoint` est un sous-type du $'a$ de `equalscolored_point`
(contravariance du domaine)
C'est-à-dire `point` est un sous-type de `colored_point`
- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Impact des « self-types » sur l'héritage et le sous-typage

Héritage avec un « self-type »

- Donc `colored_point` est un sous-type de `point` si et seulement si `point` est un sous-type de `colored_point` !
- C'est possible uniquement si `point` et `colored_point` sont égaux.
- Mais `colored_point` possède une méthode en plus (`get_color`).
- Donc `colored_point` n'est pas un sous-type de `point`.

Conclusion

- Sous-classe \nRightarrow sous-type !
- On avait déjà vu : sous-type \nRightarrow sous-classe.
- Donc, sous-classe et sous-type ne sont pas liées en OCaml.

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe

- Soit une sous-classe B d'une classe A . Le type de toute instance de B est une instance du type polymorphe $\#A$ (type ouvert).
- Ainsi, toute fonction ou classe paramétrée par $\#A$ pourra prendre indifféremment des instances de A ou de B .
- La relation entre sous-classe et super-classe repose donc plus sur le polymorphisme paramétrique que sur le polymorphisme d'inclusion !

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe

- Soit une sous-classe B d'une classe A . Le type de toute instance de B est une instance du type polymorphe $\#A$ (type ouvert).
- Ainsi, toute fonction ou classe paramétrée par $\#A$ pourra prendre indifféremment des instances de A ou de B .
- La relation entre sous-classe et super-classe repose donc plus sur le polymorphisme paramétrique que sur le polymorphisme d'inclusion !

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe

- Soit une sous-classe B d'une classe A . Le type de toute instance de B est une instance du type polymorphe $\#A$ (type ouvert).
- Ainsi, toute fonction ou classe paramétrée par $\#A$ pourra prendre indifféremment des instances de A ou de B .
- La relation entre sous-classe et super-classe repose donc plus sur le polymorphisme paramétrique que sur le polymorphisme d'inclusion !

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (fonctions paramétrées)

```
# let get_x (p : #point) = let (x, _) = p#get in x;;  
val get_x : #point → int = <fun>  
# get_x p;;  
- : int = 1  
# get_x cp;;  
- : int = 1
```


Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (fonctions paramétrées)

```
# let get_x (p : #point) = let (x, _) = p#get in x;;  
val get_x : #point → int = <fun>  
# get_x p;;  
- : int = 1  
# get_x cp;;  
- : int = 1
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (fonctions paramétrées)

```
# let get_x (p : #point) = let (x, _) = p#get in x;;  
val get_x : #point → int = <fun>  
# get_x p;;  
- : int = 1  
# get_x cp;;  
- : int = 1
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (fonctions paramétrées)

```
# let get_x (p : #point) = let (x, _) = p#get in x;;  
val get_x : #point → int = <fun>  
# get_x p;;  
- : int = 1  
# get_x cp;;  
- : int = 1
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (fonctions paramétrées)

```
# let get_x (p : #point) = let (x, _) = p#get in x;;  
val get_x : #point → int = <fun>  
# get_x p;;  
- : int = 1  
# get_x cp;;  
- : int = 1
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (fonctions paramétrées)

```
# let get_x (p : #point) = let (x, _) = p#get in x;;  
val get_x : #point → int = <fun>  
# get_x p;;  
- : int = 1  
# get_x cp;;  
- : int = 1
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# class ['a] point_list =  
object  
  constraint 'a = #point  
  val mutable content = []  
  method add (a : 'a) = content <- a :: content  
  method get = content  
end;;  
  
class ['a] point_list :  
  object  
    constraint 'a = #point  
    val mutable content : 'a list  
    method add : 'a → unit  
    method get : 'a list  
  end
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# class ['a] point_list =  
object  
  constraint 'a = #point  
  val mutable content = []  
  method add (a : 'a) = content <- a :: content  
  method get = content  
end;;  
class ['a] point_list :  
  object  
    constraint 'a = #point  
    val mutable content : 'a list  
    method add : 'a → unit  
    method get : 'a list  
  end
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
      but an expression was expected of type point  
      The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```


Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
but an expression was expected of type point  
The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
but an expression was expected of type point  
The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
      but an expression was expected of type point  
      The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;
```

*Error: This expression has type colored_point
but an expression was expected of type point
The second object type has no method get_color*

```
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
but an expression was expected of type point  
The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
but an expression was expected of type point  
The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let pl = new point_list;;  
val pl : _#point point_list = <obj>  
# pl#add p;;  
- : unit = ()  
# pl#add cp;;  
Error: This expression has type colored_point  
but an expression was expected of type point  
The second object type has no method get_color  
# pl;;  
- : point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let cpl = new point_list;;  
val cpl : _#point point_list = <obj>  
# cpl#add cp;;  
- : unit = ()  
# cpl;;  
- : colored_point point_list = <obj>
```


Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let cpl = new point_list;;  
val cpl : _#point point_list = <obj>  
# cpl#add cp;;  
- : unit = ()  
# cpl;;  
- : colored_point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let cpl = new point_list;;  
val cpl : _#point point_list = <obj>  
# cpl#add cp;;  
- : unit = ()  
# cpl;;  
- : colored_point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let cpl = new point_list;;  
val cpl : _#point point_list = <obj>  
# cpl#add cp;;  
- : unit = ()  
# cpl;;  
- : colored_point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let cpl = new point_list;;  
val cpl : _#point point_list = <obj>  
# cpl#add cp;;  
- : unit = ()  
# cpl;;  
- : colored_point point_list = <obj>
```

Quelle relation entre sous-classe et super-classe ?

Instance de type polymorphe (classes paramétrées)

```
# let cpl = new point_list;;  
val cpl : _#point point_list = <obj>  
# cpl#add cp;;  
- : unit = ()  
# cpl;;  
- : colored_point point_list = <obj>
```