

Objective Caml version 3.06

```
# type s = <x: <a:int; b: int>; y: int>;
type s = < x : < a : int; b : int >; y : int >

# type t = <x: <a: int> >;
type t = < x : < a : int > >

# class c_a =
object
  method a = 0
end;;
class c_a : object method a : int end

# class c_ab =
object
  inherit c_a
  method b = 1
end;;
class c_ab : object method a : int method b : int end

# class c_t =
object
  method x = new c_a
end;;
class c_t : object method x : c_a end

# class c_s =
object
  method x = new c_ab
  method y = 2
end;;
class c_s : object method x : c_ab method y : int end

# let (o_t: t) = new c_t;;
val o_t : t = <obj>

# let (o_s: s) = new c_s;;
val o_s : s = <obj>

# (o_t :> s);;
Characters 1-4:
(o_t :> s);;
  ^^^
This expression cannot be coerced to type
  s = < x : < a : int; b : int >; y : int >;
it has type t = < x : < a : int > > but is here used with type
  < x : < a : int; b : int; .. >; y : int; .. >
Only the second object type has a method y

# o_s;;
- : s = <obj>

# (o_s :> t);;
- : t = <obj>

# (o_s: s :> t);;
- : t = <obj>

# ((o_s, o_s) :> t * t);;
- : t * t = (<obj>, <obj>)

# ((o_s, o_t) :> t * t);;
- : t * t = (<obj>, <obj>)

# ((o_s, o_s) :> s * t);;
- : s * t = (<obj>, <obj>)

# (ref o_s: s ref :> t ref);;
Characters 0-25:
(ref o_s: s ref :> t ref);;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Type s ref is not a subtype of type t ref
Type s = < x : < a : int; b : int >; y : int > is not compatible with type
  t = < x : < a : int > >
Only the first object type has a method y

# (ref o_t: t ref :> s ref);;
```

Characters 0-25:

```
(ref o_t: t ref := s ref);;
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Type t ref is not a subtype of type s ref

Type t = < x : < a : int > > is not compatible with type

```
s = < x : < a : int; b : int >; y : int >
```

Only the second object type has a method y

```
# ([o_s] :=> t list);;
- : t list = [<obj>]
```

```
# let produce_s = function () -> o_s;;
val produce_s : unit -> s = <fun>
```

```
# let produce_t = function () -> o_t;;
val produce_t : unit -> t = <fun>
```

```
# (produce_s :=> unit -> t);;
- : unit -> t = <fun>
```

```
# (produce_t :=> unit -> s);;
```

Characters 1-10:

```
(produce_t :=> unit -> s);;
^^^^^^^^^^^^
```

This expression cannot be coerced to type unit -> s; it has type unit -> t but is here used with type

```
unit -> (< x : < a : int; b : int; .. >; y : int; .. > as 'a)
```

Type t = < x : < a : int > > is not compatible with type 'a

Only the second object type has a method y

```
# let consume_s = function (x: s) -> ();;
val consume_s : s -> unit = <fun>
```

```
# let consume_t = function (x: t) -> ();;
val consume_t : t -> unit = <fun>
```

```
# (consume_s :=> t -> unit);;
```

Characters 1-10:

```
(consume_s :=> t -> unit);;
^^^^^^^^^^^^
```

This expression cannot be coerced to type t -> unit; it has type s -> unit but is here used with type t -> unit

Type s = < x : < a : int; b : int >; y : int > is not compatible with type t = < x : < a : int > >

Only the first object type has a method y.

This simple coercion was not fully general. Consider using a double coercion.

```
# (consume_t :=> s -> unit);;
```

Characters 1-10:

```
(consume_t :=> s -> unit);;
^^^^^^^^^^^^
```

This expression cannot be coerced to type s -> unit; it has type t -> unit but is here used with type s -> unit

Type t = < x : < a : int > > is not compatible with type

```
s = < x : < a : int; b : int >; y : int >
```

Only the second object type has a method y.

This simple coercion was not fully general. Consider using a double coercion.

```
# (consume_t: t -> unit :=> s -> unit);;
- : s -> unit = <fun>
```

```
# (consume_s: s -> unit :=> t -> unit);;
```

Characters 0-35:

```
(consume_s: s -> unit :=> t -> unit);;
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Type s -> unit is not a subtype of type t -> unit

Type t = < x : < a : int > > is not a subtype of type

```
s = < x : < a : int; b : int >; y : int >
```

```
# let f (x: t) = o_s;;
val f : t -> s = <fun>
```

```
# (f :=> s -> t);;
```

Characters 1-2:

```
(f :=> s -> t);;
^
```

This expression cannot be coerced to type s -> t; it has type t -> s but is here used with type s -> < x : < a : int; .. >; .. >

Type `t = < x : < a : int > >` is not compatible with type

`s = < x : < a : int; b : int >; y : int >`

Only the second object type has a method `y`.

This simple coercion was not fully general. Consider using a double coercion.

```
# (f: t -> s :> s -> t);;
- : s -> t = <fun>
```

```
# let g (x: s) = o_t;;
val g : s -> t = <fun>
```

```
# (g: s -> t :> t -> s);;
```

Characters 0-21:

```
(g: s -> t :> t -> s);;
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Type `s -> t` is not a subtype of type `t -> s`

Type `t = < x : < a : int > >` is not a subtype of type

`s = < x : < a : int; b : int >; y : int >`

```
# type u = {a: s};;
type u = { a : s; }
```

```
# let r_u = {a = o_s};;
val r_u : u = {a = <obj>}
```

```
# type v = {a: t};;
type v = { a : t; }
```

```
# (r_u :> v);;
```

Characters 1-4:

```
(r_u :> v);;
^^^
```

This expression cannot be coerced to type `v`; it has type `u` but is here used with type `v`

```
# (r_u: u :> v);;
```

Characters 0-13:

```
(r_u: u :> v);;
^^^^^^^^^^^^^^^^^^^^
```

Type `u` is not a subtype of type `v`

```
# type u' = {a: s};;
type u' = { a : s; }
```

```
# (r_u :> u');;
```

Characters 1-4:

```
(r_u :> u');;
^^^
```

This expression cannot be coerced to type `u'`; it has type `u` but is here used with type `u'`

```
# r_u.a;;
```

Characters 0-3:

```
r_u.a;;
^^^
```

This expression has type `u` but is here used with type `u'`

```
# let a = Array.create 10 o_s;;
```

```
val a : s array =
```

```
[|<obj>; <obj>; <obj>; <obj>; <obj>; <obj>; <obj>; <obj>; <obj>; <obj>|]
```

```
# (a :> t array);;
```

Characters 1-2:

```
(a :> t array);;
^
```

This expression cannot be coerced to type `t array`; it has type `s array` but is here used with type `t array`

Type `s = < x : < a : int; b : int >; y : int >` is not compatible with type

`t = < x : < a : int > >`

Only the first object type has a method `y`

```
# type u = A of s;;
type u = A of s
```

```
# let a_u = A o_s;;
val a_u : u = A <obj>
```

```
# type v = A of t;;
```

```

type v = A of t

# (a_u :> v);;
Characters 1-4:
  (a_u :> v);;
  ^^^
This expression cannot be coerced to type v; it has type u
but is here used with type v

# (a_u: u :> v);;
Characters 0-13:
  (a_u: u :> v);;
  ^^^^^^^^^^^^^^^
Type u is not a subtype of type v

# (1 :> int);;
- : int = 1

# (1: int :> int);;
- : int = 1

# type 'a producer = unit -> 'a;;
type 'a producer = unit -> 'a

# let produce_s: s producer = function () -> o_s;;
val produce_s : s producer = <fun>

# (produce_s :> t producer);;
- : t producer = <fun>

# type 'a consumer = 'a -> unit;;
type 'a consumer = 'a -> unit

# let consume_t: t consumer = function (x: t) -> ();;
val consume_t : t consumer = <fun>

# (consume_t :> s consumer);;
Characters 1-10:
  (consume_t :> s consumer);;
  ^^^^^^^^^^^
This expression cannot be coerced to type s consumer = s -> unit; it has type
  t consumer = t -> unit
but is here used with type s consumer = s -> unit
Type t = < x : < a : int > > is not compatible with type
  s = < x : < a : int; b : int >; y : int >
Only the second object type has a method y.
This simple coercion was not fully general. Consider using a double coercion.

# (consume_t: t consumer :> s consumer);;
- : s consumer = <fun>

# type 'a tree =
  Leaf of 'a
  | Node of 'a tree * 'a tree;;
  type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree

# let x: s tree = Leaf o_s;;
val x : s tree = Leaf <obj>

# (x :> t tree);;
- : t tree = Leaf <obj>

# type 'a stream =
  Cons of 'a * (unit -> 'a stream);;
  type 'a stream = Cons of 'a * (unit -> 'a stream)

# let rec x: s stream = Cons (o_s, function () -> x);;
val x : s stream = Cons (<obj>, <fun>)

# (x :> t stream);;
- : t stream = Cons (<obj>, <fun>)

# (x :> 'a stream);;
- : s stream = Cons (<obj>, <fun>)

#

```