プログラミング言語 5
OCamlのオブジェクト指向型システム
Programming Languages 5
Object-Oriented Type System of OCaml

田浦

目次

1. OCaml のオブジェクト：型苦しい話以前/ Objects in OCaml
2. OCaml のオブジェクト：部分型/ OCaml objects：subtypes
3. (ついに) 継承について / about inheritance (finally)
4. (補足) OCaml の配列とレコード/ (Appendix) arrays and records in OCaml

Contents

1. OCaml のオブジェクト：型苦しい話以前/ Objects in OCaml
2. OCaml のオブジェクト：部分型/ OCaml objects：subtypes
3. (ついに) 継承について / about inheritance (finally)
4. (補足) OCaml の配列とレコード/ (Appendix) arrays and records in OCaml

OCamlのオブジェクト指向の精神

- オブジェクト型はオブジェクトの持つメソッドを規定するための構造で，クラスとは独立なもの
- 実際オブジェクトはクラスを定義しなくても作れる
- 部分型関係はその構造 (どんなメソッドを持つか，その型はなにか) から決まり，ある型が「どう定義 (特に, 継承) されたか」にはよらない
- クラスの主目的は継承による再利用
- 継承と部分型は直交．継承したクラスが元の部分型になるかどうかは，あくまで両者の構造から決まる

Philosophy of the OCaml Object Type System

- object types represent the methods objects have and *are independent from classes*
- in fact, objects can be created without defining classes
- subtype relationships are determined by their structures (methods they have and their types), not by *"how they are defined (derived)"*
- classes do exist, for reusing them by inheritance
- inheritance and subtype are orthogonal. *whether a derived type becomes a subtype of the parent is determined by their structure*

オブジェクトを作る (クラスは無くても)

- bb と to_svg というメソッドを持つオブジェクト (クラスなしに作れる)

```
# let r = object method bb = (10,10,30,30) method to_svg = "..." end;;
val r : < bb :  int * int * int * int; to_svg :  string > = <obj>
```

- object 式のひな形:

```
object
  method f = 式
  method g x = 式
    ...
end
```

- < ... >が「オブジェクト型」
- $m : t$ が「メソッド m を持ちその型が t」の意味
- メソッド呼び出しは，オブジェクト#メソッド名 引数 ...

```
# r#to_svg
- : string =
"<rect x=\"10\" y=\"10\" width=\"30\" height=\"30\"/>"
```

Creating an object (without classes)

- an object that has methods `bb` and `to_svg` (you can create it without a class)

```
# let r = object method bb = (10,10,30,30) method to_svg = "..." end;;
val r : < bb :  int * int * int * int; to_svg :  string > = <obj>
```

- syntax of object expressions:

```
object
  method f = expr
  method g x = expr
    ...
end
```

- `< ... >` represents "object types"
- $m : t$ means "it has method m of type t"
- メソッド呼び出しは，オブジェクト#メソッド名 引数 ...
- syntax of method calls *object*#*method arg* ...

```
# r#to_svg
- : string =
"<rect x=\"10\" y=\"10\" width=\"30\" height=\"30\"/>"
```

オブジェクトを作る (クラスは無くても)

- 「オブジェクトを返す関数」を書けば ≈ コンストラクタ

```
# let mk_rect x y width height = object
    method bb = (x, y, x+width, y+height)
    method to_svg = "..."
end;;
val mk_rect :
  int -> int -> int -> int -> < bb : int * int * int * int; to_svg :
          string > =
  <fun>
```

- クラスみたいなものがこれでできた
- オブジェクトを作りたいだけならクラスはいらない

Creating an object (without classes)

- write any function that returns an object and it is $\approx$ a constructor

```
# let mk_rect x y width height = object
    method bb = (x, y, x+width, y+height)
    method to_svg = "..."
end;;
val mk_rect :
  int -> int -> int -> int -> < bb : int * int * int * int; to_svg :
          string > =
  <fun>
```

- we already have something like a class, don't we?
- classes are unnecessary if they are just for creating objects

クラスはあるんですか，ないんですか?

- クラスはあります!

```
# class rect x y width height = object
  method bb = (x,y,x+width,y+height)
  method to_svg = "..."
end;;
```

- コンストラクタを普通の関数で書くのとあまりかわらない
- クラスを作ったらそのオブジェクトは，new という特別の構文で作る

```
# new rect 0 0 100 100
```

- つまり new クラス名 がオブジェクトを生む関数になる

```
# new rect ;;
- : int -> int -> int -> int -> rect = <fun>
```

- クラスの真価は継承するときに生まれる (後述)

So do we have classes?

- Yes, we do!

```
# class rect x y width height = object
  method bb = (x,y,x+width,y+height)
  method to_svg = "..."
end;;
```

- the syntax is not very different from defining constructors with ordinary functions
- when you define a class, you make an object of the class with `new` syntax

```
# new rect 0 0 100 100
```

- that is `new` *class-name* is a function to construct an object

```
# new rect ;;
- : int -> int -> int -> int -> rect = <fun>
```

- the true value of classes is in inheritance (discussed later)

型パラメータを持つクラスに関する注意

- 復習: パラメトリックな多相関数は OCaml が自動的に推論

```
1 # let f x = x ;;
2 val f : 'a -> 'a = <fun>
```

- オブジェクトを返す関数も同様

```
1 # let cell x = object method get = x end ;;
2 val cell : 'a -> < get : 'a > = <fun>
```

- 結果的にこれだけで「ジェネリックなクラス」ができる

```
1 # let c0 = cell 3;;       (* < get : int > *)
2 # let c1 = cell "hello"; (* < get : string > *)
```

A caveat for classes with type parameters

- review: OCaml automatically infers functions types having type parameters

```
# let f x = x ;;
val f : 'a -> 'a = <fun>
```

- functions returning objects are no different

```
# let cell x = object method get = x end ;;
val cell : 'a -> < get : 'a > = <fun>
```

- we effectively get "generic classes", with no new mechanism in OCaml

```
# let c0 = cell 3;;       (* < get : int > *)
# let c1 = cell "hello"; (* < get : string > *)
```

型パラメータを持つクラスに関する注意

- だが同じことをクラスで書くとなぜかエラーになる

```
# class cell x = object method get = x end ;;
Characters 6-40:
  class cell x = object method get = x end ;;
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Some type variables are unbound in this type:
         class cell : 'a -> object method get : 'a end
       The method get has type 'a where 'a is unbound
```

- 規則: クラスのコンストラクタ関数 (この例での `new cell`) が多相型になる (この例では`'a -> < get : 'a>`) 場合，それを明示的にクラスの型パラメータに指定する必要がある

```
# class ['a] cell (x : 'a) = object method get = x end ;;
```

注：`(x : 'a)` は型を明示的に指定する構文

A caveat with classes with type parameters

- the same example will raise an error if written using classes, however (just an implementation glitch?)

```
# class cell x = object method get = x end ;;
Characters 6-40:
  class cell x = object method get = x end ;;
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Some type variables are unbound in this type:
         class cell : 'a -> object method get : 'a end
       The method get has type 'a where 'a is unbound
```

- rule: when a class constructor (`new cell` in this example) becomes polymorphic (`'a -> < get : 'a>` in this example), you must indicate that with a type parameter

```
# class ['a] cell (x : 'a) = object method get = x end ;;
```

note : `(x : 'a)` is a syntax to explicitly specify a type of a variable

色々なオブジェクトに多相的に働く関数

- to_svg というメソッドだけを使う関数を書けば，その関数は「to_svg を持つオブジェクトなら何でも OK」な関数になる

```
# let svg_of s = s#to_svg ^ "\n";;
val svg_of : < to_svg : string; .. > -> string = <fun>
```

- < to_svg : string; .. >は，string を返すメソッド to_svg を持ち，他にも持っているかもしれないオブジェクトの型
- そのようなオブジェクトならなんでも svg_of に適用可能

```
# class circle cx cy radius = object
    method to_svg = "..."
    method radius = radius
end;;
# svg_of (new rect 30 30 100 100);;
# svg_of (new circle 100 100 50);;
```

- circle の持つメソッドは (説明の都合上あえて) rect と同一でないことに注意

Polymorphic functions applicable to various objects

- if you write a function that uses only `to_svg` method, it can accept *"any object having* `to_svg` *method"*

```
# let svg_of s = s#to_svg ^ "\n";;
val svg_of : < to_svg : string; .. > -> string = <fun>
```

- `< to_svg : string; .. >` represents the type of objects having `to_svg` method returning `string`, *possibly with others*
- we can `svg_of` to any such objects

```
# class circle cx cy radius = object
    method to_svg = "..."
    method radius = radius
end;;
# svg_of (new rect 30 30 100 100);;
# svg_of (new circle 100 100 50);;
```

- note that methods of `circle` are not the same as those of `rect`

メソッドじゃないフィールドはあるんですか?

- フィールドは，あります! 文法: `val` 変数名 = 値

```
# let o = object
  val x = 100
  method add y = y + x
end ;;
```

- フィールドはその型には現れない

```
# o ;;
- : < add : int -> int > = <obj>
```

- フィールドはオブジェクトの外からアクセスできない

```
# o#x ;;
Characters 0-1:
  o#x ;;
  ^
Error: This expression has type < add : int -> int >
       It has no method x
```

- 外からアクセスしたければ (0 引数の) メソッドにすれば良い
- 他の OCaml の変数同様，(デフォルトでは) immutable

We have methods, but do we have fields?

- Yes we do! syntax: `val` *name* = *expr*

```
# let o = object
  val x = 100
  method add y = y + x
end ;;
```

- fields are not revealed as the type of the object

```
# o ;;
- : < add : int -> int > = <obj>
```

- fields are not directly accessible from outside the object

```
# o#x ;;
Characters 0-1:
  o#x ;;
  ^
Error: This expression has type < add : int -> int >
       It has no method x
```

- if you want to make it externally accessible, make it a method (with zero input parameters)
- like other OCaml variables, fields are, by default, immutable

フィールド (val) vs. 引数なしのメソッド

```
1  object
2    val x = e
3    ...
4  end
```

は, 以下と似ている

```
1  object
2    method x = e
3    ...
4  end
```

違い:

- フィールドはオブジェクト外部からはアクセスできず, 型にも現れない
- フィールドは mutable にできる (後述)
- フィールドは, 関数的更新もできる (後述)

Fields (`val`) vs. methods with no arguments

```
1  object
2    val x = e
3    ...
4  end
```

is similar to:

```
1  object
2    method x = e
3    ...
4  end
```

how they differ:

- fields are not accessible from outside the object and are not part of types
- fields can be mutable (later)
- fields can be functionally updated (later)

mutable なフィールド

- mutable なフィールドはあります!
- 文法: `val mutable` 変数名 = 値
- 更新は `<-` を用いる

```
1  # class accumulator x = object
2    val mutable x = x
3    method inc y = x <- x + y; x
4  end ;;
5  # let a = new accumulator 3 ;;
6  # a#inc 4;;
7  - : int = 7
8  # a#inc 10;;
9  - : int = 17
```

mutable fields

- Yes, we do have mutable fields!
- syntax: `val mutable` *name* = *expr*
- updates use `<-`

```
1  # class accumulator x = object
2    val mutable x = x
3    method inc y = x <- x + y; x
4  end ;;
5  # let a = new accumulator 3 ;;
6  # a#inc 4;;
7  - : int = 7
8  # a#inc 10;;
9  - : int = 17
```

オブジェクトのコピー, および関数的な更新

- object 式中では,

  {< フィールド = 式; ...; フィールド = 式 >}

  で,「そのオブジェクトのフィールドを変更した (新しい) オブジェクト」を意味する

```
# let o = object
  val x = 1
  method inc = {< x = x + 1 >}
  method get = x
end ;;
val o :  < get :  int; inc :  'a > as 'a = <obj>
# let p = o#inc ;;
val p : < get : int; inc : 'a > as 'a = <obj>
# o#get ;;
- : int = 1
# p#get ;;
- : int = 2
```

- とくに, {< >} は「自分のコピー」になる

Copy objects and functional updates

- inside an object expression:

  $\{<\ field\ =\ expr;\ \ldots;\ field\ =\ expr\ >\}$

  means a (new) object with fields updated as specified *(functional updates)*

```
# let o = object
  val x = 1
  method inc = {< x = x + 1 >}
  method get = x
end ;;
val o :  < get :  int; inc :  'a > as 'a = <obj>
# let p = o#inc ;;
val p : < get : int; inc : 'a > as 'a = <obj>
# o#get ;;
- : int = 1
# p#get ;;
- : int = 2
```

- in particular, $\{<\ >\}$ is a copy of itself

オブジェクトのコピー (関数的な更新) と再帰的な型

- 関数的な更新式は, 「自分と同じ型」のオブジェクトを作るために使える. 例:

```
1 class c = object method copy = {< >} end ;;
```

- 以下と何が違うのか?

```
1 class c = object method copy = new c end ;;
```

- c が継承されたときに違いが生まれる. c を継承して d を作った場合:
  - {< >} は, d においては型 d になる
  - new c は, あくまで c (当然)
- 柔軟に継承可能な再帰的なデータを定義する強力な武器になる

Copy objects (functional updates) and recursive types

- functional updates can be used to create an object of "the same type with itself." ex:

```
1 class c = object method copy = {< >} end ;;
```

- any different from this?

```
1 class c = object method copy = new c end ;;
```

- the difference arises when `c` is extended. when `c` is extended to derive `d`:
  - `{< >}` is of type `d` in `d`
  - `new c`, obviously, remains of type `c`
- it is a powerful weapon to define *a recursive data type that is flexibly inheritable*

self

- object(s) ... end とすることで，定義されているオブジェクト自身を s で参照できる
  - 典型的には，自分の持つ他のメソッドを呼び出すときに用いる

```
1 class cell x = object(s)
2   method get = x + 1
3   method get2 = s#get + s#get
4 end ;;
```

- object($s : 's$) ... end とすることで，$'s$ がその型を参照する
  - 「自分と同じ型」をメソッド (フィールド) で明示可能

```
1 class cell x = object(s : 's)
2   method get = x + 1
3   method cmp (o : 's) = s#get = o#get
4 end ;;
```

  - cmp (o:cell) とするのとの違いは? ⇒ やはり cell が継承されたときに違いが生まれる
    - ★ cmp (o : cell) : 継承されても cmp は cell を受け取る
    - ★ cmp (o : 's) : 継承されたら新しいクラスを受け取る

self

- `object(`s`) ... end` allows you to reference the object being defined as s
  - typically used to call a method of itself

```
1 class cell x = object(s)
2   method get = x + 1
3   method get2 = s#get + s#get
4 end ;;
```

- `object(`$s : 's$`) ... end` allows you to refer to its type
  - allow methods (or fields) referencing "the same type as itself"

```
1 class cell x = object(s : 's)
2   method get = x + 1
3   method cmp (o : 's) = s#get = o#get
4 end ;;
```

  - any different from `cmp (o:cell)` ? $\Rightarrow$ again, a difference arises when `cell` is inherited
    - ★ `cmp (o:cell)`: continues to receive `cell` in the child classes
    - ★ `cmp (o:'s)`: receives the new (child) type in the child classes

OCamlのオブジェクトまとめ(1)

- object 式:

```
1 object
2   method f x = ···
3   method g x y = ···
4 end
```

- オブジェクト式の型: < f : ...; g : ... > 「こういうメソッドを持つオブジェクト」の型
- フィールド

```
1 object
2   val          a = ···
3   val mutable b = ···
4   ···
5 end
```

Summary of OCaml objects (1)

- object expression:

```
1  object
2    method f x = ···
3    method g x y = ···
4  end
```

- type of object expressions: `<`f `:` $\cdots$ `;` g `:` $\cdots$ `>` type of "objects having such and such methods of such types"
- fields (immutable by default; can be labeled `mutable`)

```
1  object
2    val         a = ···
3    val mutable b = ···
4    ···
5  end
```

OCamlのオブジェクトまとめ (2)

- 自分 (self) とその型

```
1 object(s : 's) = ···
```

- 破壊的更新 (<-)

```
1 object
2   val mutable b = ...
3   method f x = b <- b + x
4 end
```

自分の b を変更

- 関数的更新 ({< >})

```
1 object
2   val b = ...
3   method f x = {< b = b + x >}
4 end
```

b を b + x に変更した (残りは自分と同じ) 新しいオブジェクト

Summary of OCaml objects (2)

- self object and its type

```
1 object(s : 's) = ···
```

- destructive updates (<-)

```
1 object
2   val mutable b = ...
3   method f x = b <- b + x
4 end
```

update `b` of itself

- functional updates ({< >})

```
1 object
2   val b = ...
3   method f x = {< b = b + x >}
4 end
```

a new object with `b` being `b + x` (and other fields being intact)

OCamlのオブジェクトまとめ(3)

- class

```
1 class C x y ··· =
2   object ... end
```

  - 右辺のオブジェクトの型に C という名前をつける
  - new $C \approx$ で以下の C'

```
1 let C' x y ··· =
2   object ... end
```

- class はなくてもオブジェクトおよびその型はある

Summary of OCaml objects (3)

- class

```
1 class C x y ··· =
2   object ... end
```

  - name the type of right-hand side C
  - "`new` C" $\approx C'$ defined below

```
1 let C' x y ··· =
2   object ... end
```

- there are objects and their types before classes

Contents

1. OCaml のオブジェクト：型苦しい話以前/ Objects in OCaml
2. OCaml のオブジェクト：部分型/ OCaml objects：subtypes
3. (ついに) 継承について / about inheritance (finally)
4. (補足) OCaml の配列とレコード/ (Appendix) arrays and records in OCaml

型強制で静的な型を指示する

- rect は `to_svg` と `side`, circle は `to_svg` と `radius` メソッドを持つとする
- 型が同一でないオブジェクトをリストに混ぜると「そのままでは」静的型エラーになる

```
# [ new circle 0 0 10 ; new rect 0 0 10 10 ];;
Characters 22-40:
  [ new circle 0 0 10 ; new rect 0 0 10 10 ];;
                        ^^^^^^^^^^^^^^^^^^
Error: This expression has type rect but an expression was expected of
         type
         circle
       The second object type has no method radius
```

- 複数の型が混ざったリスト (配列, etc.) をどう作れば良い?

Specifying a static type by type coercion

- say `rect` has `to_svg` and `side` methods; and `circle` `to_svg` and `radius` methods
- putting elements of different static types in a single list will raise a static type error

```
# [ new circle 0 0 10 ; new rect 0 0 10 10 ];;
Characters 22-40:
  [ new circle 0 0 10 ; new rect 0 0 10 10 ];;
                        ^^^^^^^^^^^^^^^^^^
Error: This expression has type rect but an expression was expected of
         type
         circle
       The second object type has no method radius
```

- how to make such heterogeneous lists (arrays, etc.)?

型強制で静的な型を指示する

- 一方上記のリストを，「to_svg を持つオブジェクトのリスト」として扱うことは妥当
- 「rect も circle も，< to_svg : string >型です (side や radius は忘れて)」と言いたい
- それを明示するために，「型強制 (coercion)」がある
  - 文法: (式 :> 型)

```
1 # [ (new circle 0 0 10  :> < to_svg :  string >);
2     (new rect 0 0 10 10 :> < to_svg :  string >) ];;
3 - : < to_svg : string > list = [<obj>; <obj>]
```

- C/C++や Java のキャストと似てますが...

Specifying a static type by type coercion

- it is reasonable to treat the above list as "a list of objects having `to_svg`"
- we wish to say "both `circle` and `rect` have type `< to_svg : string >` (and forget `radius` or `side`)"
- *"type coercion"* just does that
  - syntax: *(expression :> type)*

```
# [ (new circle 0 0 10  :> < to_svg :  string >);
    (new rect 0 0 10 10 :> < to_svg :  string >) ];;
- : < to_svg : string > list = [<obj>; <obj>]
```

- somewhat similar to the type cast as you know it in C/C++ and Java ...

OCamlの型強制 $\neq$ キャスト

- 注: OCamlの型強制 ($(e$ `:>` $t)$) は，e の静的な型が t の部分型である場合のみ使える (upcast)
- C/C++の cast, Java の downcast とは違うもの
- 毎回 (e `:> < to_svg : string >`) と書くのは長いので，名前を付けたければつけられる

```
1 # type svg = < to_svg : string >;;
2 type svg = < to_svg : string >
3 # [ (new circle 0 0 10 :> svg); (new rect 0 0 10 10 :> svg) ];;
4 - : svg list = [<obj>; <obj>]
```

Type coercion of OCaml ≠ the cast as you know it

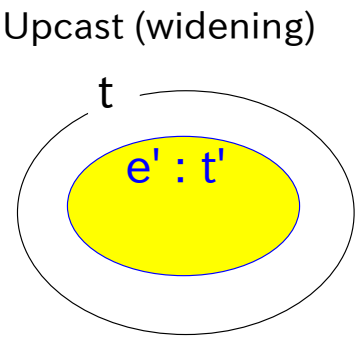
- note: type coercion of OCaml (`(e :> t)`) is valid only when the static type of e is a subtype of t (upcast)
- unlike the cast in C/C++ or the downcast in Java
- you can give a name to a type, to avoid verbosely writing (`e :> < to_svg : string >`) every time

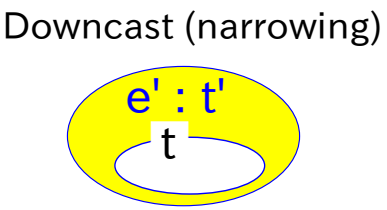
```
1 # type svg =  < to_svg : string >;;
2 type svg = < to_svg : string >
3 # [ (new circle 0 0 10 :> svg); (new rect 0 0 10 10 :> svg) ];;
4 - : svg list = [<obj>; <obj>]
```

Upcast (widening) vs. downcast (narrowing)

式 e' を型 t にキャストするといっても以下の 2 つはだいぶ違う

- Upcast (Widening)
  - e' の自然な型 $t' \leq t$ (「上位型 (supertype) へキャスト」)
  - e' は「もともと」t 型でもある．$\rightarrow$ これを許したせいで実行時に型エラーが起きることはない)
  - 主に静的な型検査を成功させるための都合で導入される
- Downcast (Narrowing)
  - $t \leq t'$ (= 「部分型 (subtype) へのキャスト」)
  - 実行時に，e' は t(またはその部分型) ではないかもしれない
  - 「実行時に型を検査するための構文」として導入される (エラー時の挙動は色々)

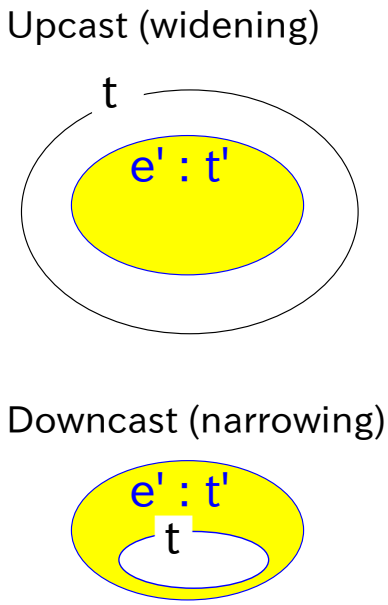




Upcast (widening) vs. downcast (narrowing)

there are two kinds of casting expression e' to type t

- upcast (widening)
  - the natural type of $e' : t' \leq t$ ("casting to supertype")
  - e' *is* of type t to begin with $\rightarrow$ it never causes a type error at runtime
  - the main purpose is simply to help a static type checker pass (already type safe) programs
- downcast (narrowing)
  - $t \leq t'$ (= "casting to subtype")
  - e' may not be of type t (or its subtype) at runtime
  - the purpose is to *type check at runtime* (behavior upon an error varies)



Downcast (Narrowing) 構文とその挙動

- Java: キャスト構文 $(t)e'$
- C++:
  - 動的キャスト `dynamic_cast<`t`>(`e'`)`
  - 通常のキャスト $(t)e'$
- Eiffel: 試行代入 v `:=` t `?=` e'
- OCaml: ありません

エラー時 (e' が t の部分型でなかった時) の挙動

Java キャスト	例外発生
C++ 動的キャスト	null ポインタ
C++ キャスト	未定義
Eiffel 試行代入	null ポインタ

The syntax and behavior of downcast (narrowing)

- Java: cast expression $(t)e'$
- C++:
  - dynamic cast `dynamic_cast<`t`>(`e'`)`
  - traditional cast ($(t)e'$), static cast, reinterpret cast, const cast
- Eiffel: assignment attempt v `:` t `?=` e'
- OCaml: none

behavior upon error (when e' is not of type t or its subtype)

Java cast	exception
C++ dynamic cast	null pointer
C++ traditional cast	undefined
Eiffel assignment attempt	null pointer

OCamlの部分型を調べる

- Coercion 式

$$(e' :> t)$$

  が静的型検査を通るか否かで, OCaml が e' の自然な型 (t' とする) を t の部分型と思っているかが判定できる
- 例

```
1 # class a = object method x = 0 method y = 1 end ;;
2 # class b = object method x = 0 end ;;
3 # (new a :> b) ;;  (* OK a <= b *)
4 # ([ new a ] :> b list) ;;  (* OK a list <= b list *)
5 # ...
```

- OCaml は Eiffel や Java と同じ間違いをしていないかも, 確かめられる (cf. 演習)
  - (a の配列式 :> b の配列型) ?
  - (int から a の関数式 :> int から b の関数型) ?
  - (a から int の関数式 :> b から int の関数型) ?
  - ...

Checking OCaml subtype relationship

- by checking if a coercion expression

$$(e' :> t)$$

  passes the static type check, you can learn if OCaml thinks the (naturally infered) type of e' (say t') is a subtype of t
- ex.

```
1 # class a = object method x = 0 method y = 1 end ;;
2 # class b = object method x = 0 end ;;
3 # (new a :> b) ;;  (* OK a <= b *)
4 # ([ new a ] :> b list) ;;  (* OK a list <= b list *)
5 # ...
```

- you can check if OCaml makes the same mistake as Eiffel or Java (see exercise)
  - (array expression of `a :>` array type of `b)` ?
  - (function of int to `a :>` function type `int` to `b)` ?
  - (function of `a` to `int :>` function type `b` to `int)` ?
  - ...

OCamlの部分型

$\alpha' \leq \alpha$, $\beta' \leq \beta$ とする

- α' `list` $\leq \alpha$ `list`
- α' `array` $\not\leq \alpha$ `array`
- $\alpha \to \beta' \leq \alpha \to \beta$
- $\alpha \to \beta \leq \alpha' \to \beta$
- (特に, $\alpha' \to \beta \not\leq \alpha \to \beta$)
- `<` `get` `:` $\alpha \to \beta'$ `>` $\leq$ `<` `get` `:` $\alpha \to \beta$ `>`
- `<` `set` `:` $\alpha \to \beta$ `>` $\leq$ `<` `set` `:` $\alpha' \to \beta$ `>`

これまでに述べたルールと完全に一致

Subtypes in OCaml

Assume $\alpha' \leq \alpha$, $\beta' \leq \beta$

- α' `list` $\leq \alpha$ `list`
- α' `array` $\not\leq \alpha$ `array`
- $\alpha \to \beta' \leq \alpha \to \beta$
- $\alpha \to \beta \leq \alpha' \to \beta$
- (in particular, $\alpha' \to \beta \not\leq \alpha \to \beta$)
- `< get :` $\alpha \to \beta'$ `>` $\leq$ `< get :` $\alpha \to \beta$ `>`
- `< set :` $\alpha \to \beta$ `>` $\leq$ `< set :` $\alpha' \to \beta$ `>`

they totally matches subtype relationships described so far

クラスの部分型関係

- 例えば,

```
1 class b = object
2   method cmp (x : b) = ()
3   method hi = "hi"
4 end
```

は

```
1 class a = object
2   method cmp (x : a) = ()
3 end
```

の部分型ではない

- メソッドを増やした (拡張した) から部分型になると思ってはいけない
- 継承したクラスについても同様に当てはまる

Subtypes between classes

- for example,

```
1 class b = object
2   method cmp (x : b) = ()
3   method hi = "hi"
4 end
```

is *not* a subtype of

```
1 class a = object
2   method cmp (x : a) = ()
3 end
```

- adding a method to (extending) a class does not automatically make a subtype of it
- it is true even when you inherit classes

再帰的なオブジェクトの型

```
1 object(s :  's)
2   method get = 3
3   method cmp (o : 's) = s#get = o#get
4 end ;;
```

の型は?

- 答え:

```
1 # object(s : 's)
2     method get = 3
3     method cmp (o : 's) = s#get = o#get
4   end ;;
5 - : < cmp :  'a -> bool; get :  int > as 'a = <obj>
```

つまり,

$$C = \texttt{< cmp : } C \texttt{ -> bool; get : int >}$$

を満たす C という型

Types of recursive objects

- what is the type of

```
1 object(s :  's)
2   method get = 3
3   method cmp (o : 's) = s#get = o#get
4 end ;;
```

?

- let's ask OCaml the answer:

```
1 # object(s : 's)
2     method get = 3
3     method cmp (o : 's) = s#get = o#get
4   end ;;
5 - : < cmp :  'a -> bool; get :  int > as 'a = <obj>
```

that is, C that satisfies:

$$C = \texttt{< cmp : } C \texttt{ -> bool; get : int >}$$

Contents

1. OCaml のオブジェクト：型苦しい話以前/ Objects in OCaml
2. OCaml のオブジェクト：部分型/ OCaml objects：subtypes
3. (ついに) 継承について / about inheritance (finally)
4. (補足) OCaml の配列とレコード/ (Appendix) arrays and records in OCaml

継承

- 文法: object ...end の中に inherit コンストラクタ式

```
# class point x y = object
    method norm2 = x * x + y * y
  end;;
```

```
# class cpoint x y (c : string) = object
    inherit point x y
    method color = c
  end;;
```

- 「inherit 句 ≈ 継承しているクラスの定義をその場にコピー」と思っておけば良い

Inheritance

- syntax: `inherit` *constructor expression* inside `object ...end`

```
# class point x y = object
    method norm2 = x * x + y * y
  end;;
```

```
# class cpoint x y (c : string) = object
    inherit point x y
    method color = c
  end;;
```

- you may consider: "`inherit` clause ≈ copy&paste the definition of the inherited class"

継承と部分型

- 継承と部分型には直接の関係はない
- 原則:
  - オブジェクトは「オブジェクト型」を持つ
  - 結果としてクラスには (そのクラスから生まれるオブジェクトの) オブジェクト型が対応する
  - オブジェクト型の部分型関係はその構造 (どんなフィールドを持ちそれらの型が何か) だけで決まる
- $\Rightarrow$ 継承した結果，部分型ができる時もあればそうでない時もある

Inheritance and subtype

- there are *no* direct relationship between inheritance and subtype
- principle:
  - an object has an "object type"
  - a class naturally corresponds to a type (of objects created by the class)
  - subtype relation between object types is *solely* determined by their structures (fields and their types)
- $\Rightarrow$ a class made by inheritance may or may not be a subtype of the base

継承と部分型

- 簡単な例

```
1 class cell x = object
2   val x = x + 1
3   method get = x + 1
4 end ;;
```

を継承 (get2 を追加)

```
1 class cell2 x = object
2   inherit cell x
3   method get2 = x + 2
4 end ;;
```

- cell2 $\leq$ cell か?

```
1 # (new cell2 10 :> cell) ;;
2 - : cell = <obj>
```

$\rightarrow$ YES

Inheritance and subtype

- a simple example:

```
1 class cell x = object
2   val x = x + 1
3   method get = x + 1
4 end ;;
```

inherit this (add `get2`)

```
1 class cell2 x = object
2   inherit cell x
3   method get2 = x + 2
4 end ;;
```

- `cell2` $\leq$ `cell`?

```
1 # (new cell2 10 :> cell) ;;
2 - : cell = <obj>
```

$\rightarrow$ YES

cell2 $\leq$ cell の理解

- 継承のことは忘れる
- それぞれの型をオブジェクト型を使って陽に書いて，部分型を判定
  - cell のオブジェクト：`< get : int >`
  - cell2 のオブジェクト：`< get : int; get2 : int >`

  たしかに，

$$< \texttt{get} : \texttt{int}; \texttt{get2} : \texttt{int} > \leq < \texttt{get} : \texttt{int} >$$

understanding `cell2` $\leq$ `cell`

- forget about inheritance
- explicitly write their object types and judge their subtype relationship
  - `cell` object : `< get : int >`
  - `cell2` object : `< get : int; get2 : int >`

  indeed,

$$< \texttt{get} : \texttt{int}; \texttt{get2} : \texttt{int} > \leq < \texttt{get} : \texttt{int} >$$

継承と部分型：自分を引数にとるメソッド

- 元クラス

```
1 class cell x = object(s : 's)
2   val x = x + 1
3   method get = x + 1
4   method cmp (o :  's) = s#get = o#get
5 end ;;
```

- 継承の仕方は全く同じ

```
1 class cell2 x = object
2   inherit cell x
3   method get2 = x + 2
4 end ;;
```

Inheritance and subtype : a method taking the type of self

- base class

```
1  class cell x = object(s : 's)
2    val x = x + 1
3    method get = x + 1
4    method cmp (o :  's) = s#get = o#get
5  end ;;
```

- the inheriting class is exactly the same

```
1  class cell2 x = object
2    inherit cell x
3    method get2 = x + 2
4  end ;;
```

継承と部分型：自分を引数にとるメソッド

- cell2 $\leq$ cell か?

```
# (new cell2 10 :> cell) ;;
Characters 0-22:
  (new cell2 10 :> cell) ;;
  ^^^^^^^^^^^^^^^^^^^^^^
Error: Type cell2 = < cmp : cell2 -> bool; get : int; get2 : int >
       is not a subtype of cell = < cmp : cell -> bool; get : int >
Type cell = < cmp : cell -> bool; get : int > is not a subtype of
  cell2 = < cmp : cell2 -> bool; get : int; get2 : int >
```

$\rightarrow$ NO

- 注: 対話的処理系なのでわかりづらいが，あくまで「静的な」型検査のエラー

Inheritance and subtype : a method taking the type of self

- `cell2` $\leq$ `cell` ?

```
# (new cell2 10 :> cell) ;;
Characters 0-22:
  (new cell2 10 :> cell) ;;
  ^^^^^^^^^^^^^^^^^^^^^^
Error: Type cell2 = < cmp : cell2 -> bool; get : int; get2 : int >
       is not a subtype of cell = < cmp : cell -> bool; get : int >
Type cell = < cmp : cell -> bool; get : int > is not a subtype of
  cell2 = < cmp : cell2 -> bool; get : int; get2 : int >
```

$\rightarrow$ NO

- note: this is a static (prior-to-execution) error, though not very clear in the interactive environment

なぜ cell2 ≤ cell ではないか?

- 関数の部分型の規則を思い出す
- cell : < cmp : cell -> bool; get : int >
- cell2 : < cmp : cell2 -> bool; get : int; get2 : int >
- $$\begin{aligned} & \texttt{cell2} \leq \texttt{cell} \\ \Rightarrow\ & \texttt{cell2} \to \texttt{bool} \leq \texttt{cell} \to \texttt{bool} \\ \Rightarrow\ & \texttt{cell} \leq \texttt{cell2} \end{aligned}$$

  だが，cell2 にのみ get2 メソッドがあるのでこれは矛盾.
  従って,

  $$\texttt{cell2} \not\leq \texttt{cell}$$

Why not `cell2` $\leq$ `cell`?

- remember subtype between function types
- `cell : < cmp : cell -> bool; get : int >`
- `cell2 : < cmp : cell2 -> bool; get : int; get2 : int >`
- $$\begin{aligned} & \texttt{cell2} \leq \texttt{cell} \\ \Rightarrow\ & \texttt{cell2} \to \texttt{bool} \leq \texttt{cell} \to \texttt{bool} \\ \Rightarrow\ & \texttt{cell} \leq \texttt{cell2} \end{aligned}$$

  contradiction as only `cell2` has `get2` method. therefore,

$$\texttt{cell2} \not\leq \texttt{cell}$$

いいかげん疲れた? 全体のまとめ

- 「オブジェクト指向を静的に型付けする」をなめてはいけない
- 「何は部分型とみなしても OK か」正しく把握するのが基本
  - 関数. $(s' \to t' \leq s \to t \iff s \leq s' \text{ and } t' \leq t)$
  - レコード. immutable な場合と mutable な場合
- 「継承」の結果出てくる型が自動的に部分型になると思ってはいけない
  - 上の規則に忠実であれば間違いを犯す心配はない

Exhausted? final summary

- "statically typing object oriented languages" is no easy matter
- solid understanding of "what is subtype of what" is the basic
  - function: $(s' \to t' \leq s \to t \iff s \leq s' \text{ and } t' \leq t)$
  - record: immutable and mutable fields
- never think "inheritance" always creates a subtype
  - no need to worry if you follow the above rules

Contents

1. OCaml のオブジェクト：型苦しい話以前/ Objects in OCaml
2. OCaml のオブジェクト：部分型/ OCaml objects：subtypes
3. (ついに) 継承について / about inheritance (finally)
4. (補足) OCaml の配列とレコード/ (Appendix) arrays and records in OCaml

OCamlについて語っていないことの補足

- 配列
- (オブジェクトではない) レコード

A few notes on what I didn't cover about OCaml

- arrays
- records (not objects)

配列

- 例

```
1 # let a = [| 1; 2; 3 |] ;;
2 val a : int array = [|1; 2; 3|]
3 # a.(0) ;;
4 - : int = 1
5 # a.(1) <- 10 ;;
6 - : unit = ()
7 # a ;;
8 - : int array = [|1; 10; 3|]
```

- OCaml Array モジュールのページを参照

Array

- ex.

```
# let a = [| 1; 2; 3 |] ;;
val a : int array = [|1; 2; 3|]
# a.(0) ;;
- : int = 1
# a.(1) <- 10 ;;
- : unit = ()
# a ;;
- : int array = [|1; 10; 3|]
```

- see OCaml Array Modules page

レコード型

- オブジェクト型，バリアントと似て非なるものにレコード型がある
- 明示的なフィールド名を持つ．≈ C の構造体

```
# type point = { x :  int; y :  int };;
type point = { x : int; y : int; }
# { x = 3; y = 4 } ;;
- : point = {x = 3; y = 4}
# p.x ;;
- : int = 3
```

Record types

- there are record types, similar to but different from object types or variant types
- (unlike variants) records have field names ≈ struct in C

```
# type point = { x :  int; y :  int };;
type point = { x : int; y : int; }
# { x = 3; y = 4 } ;;
- : point = {x = 3; y = 4}
# p.x ;;
- : int = 3
```

レコードの更新

- 破壊的更新．オブジェクト同様，フィールドを mutable にできる

```
# type mpoint = { mutable x :  int; mutable y : int; mutable z : int };;
type mpoint = { mutable x : int; mutable y : int; mutable z : int; }
# let p = { x = 3 ; y = 4; z = 5 };;
val p : mpoint = {x = 3; y = 4; z = 5}
# p.x <- 30 ;;
- : unit = ()
# p ;;
- : mpoint = {x = 30; y = 4; z = 5}
```

- 関数的更新．{ 式 with x = ...; y = ...}

```
# type point = { x : int; y : int; z : int };;
type point = { x : int; y : int; z : int; }
# let p = { x = 3 ; y = 4; z = 5 };;
val p : point = {x = 3; y = 4; z = 5}
# let q = { p with x = 30; y = 40 };;
val q : point = {x = 30; y = 40; z = 5}
# p;;
- : point = {x = 3; y = 4; z = 5}
```

Updating records

- destructive (in-place) update; fields can be made mutable (like objects)

```
# type mpoint = { mutable x :  int; mutable y : int; mutable z : int };;
type mpoint = { mutable x : int; mutable y : int; mutable z : int; }
# let p = { x = 3 ; y = 4; z = 5 };;
val p : mpoint = {x = 3; y = 4; z = 5}
# p.x <- 30 ;;
- : unit = ()
# p ;;
- : mpoint = {x = 30; y = 4; z = 5}
```

- functional updates; { *expr* with x = ...; y = ...}

```
# type point = { x : int; y : int; z : int };;
type point = { x : int; y : int; z : int; }
# let p = { x = 3 ; y = 4; z = 5 };;
val p : point = {x = 3; y = 4; z = 5}
# let q = { p with x = 30; y = 40 };;
val q : point = {x = 30; y = 40; z = 5}
# p;;
- : point = {x = 3; y = 4; z = 5}
```

フィールド名は全体で一意

- あるフィールド名を持つレコード型はひとつしか作れない (式から型を一意に推論するため)
- 同じフィールドを持つレコード型を複数定義すると, 最後に定義された型がそのフィールドを持つ

```
# type xy = { x : int ; y : int };;
type xy = { x : int; y : int; }
# type xyz = { x : int ; y : int ; z : int };;
type xyz = { x : int; y : int; z : int; }
# { x = 10 ; y = 20 };;
Characters 0-19:
  { x = 10 ; y = 20 };;
  ^^^^^^^^^^^^^^^^^^^
Error: Some record field labels are undefined:  z
```

- object 型のような，width subtyping は無し → フィールド名を見れば型が一意に決まる
- フィールド名の衝突を避けるためにはモジュールを用いる

Field names must be globally unique

- you can make only one record type with a given field name (so that a record expression can be uniquely typed)
- when you define multiple records having the same name, the one defined last owns it

```
# type xy = { x : int ; y : int };;
type xy = { x : int; y : int; }
# type xyz = { x : int ; y : int ; z : int };;
type xyz = { x : int; y : int; z : int; }
# { x = 10 ; y = 20 };;
Characters 0-19:
  { x = 10 ; y = 20 };;
  ^^^^^^^^^^^^^^^^^^^
Error: Some record field labels are undefined:  z
```

- unlike object types, there is no width subtyping between records → a single field uniquely determines the type
- to avoid conflicts of field names, use modules