13 Objects

This chapter was written by Leo White and Jason Hickey.

We've already seen several tools that OCaml provides for organizing programs, particularly modules. In addition, OCaml also supports object-oriented programming. There are objects, classes, and their associated types. In this chapter, we'll introduce you to OCaml objects and subtyping. In the next chapter, Chapter 14 (Classes), we'll introduce you to classes and inheritance.

What Is Object-Oriented Programming?

Object-oriented programming (often shortened to OOP) is a programming style that encapsulates computation and data within logical *objects*. Each object contains some data stored in *fields* and has *method* functions that can be invoked against the data within the object (also called "sending a message" to the object). The code definition behind an object is called a *class*, and objects are constructed from a class definition by calling a constructor with the data that the object will use to build itself.

There are five fundamental properties that differentiate OOP from other styles:

Abstraction The details of the implementation are hidden in the object, and the external interface is just the set of publicly accessible methods.

Dynamic lookup When a message is sent to an object, the method to be executed is determined by the implementation of the object, not by some static property of the program. In other words, different objects may react to the same message in different ways.

Subtyping If an object `a` has all the functionality of an object `b`, then we may use `a` in any context where `b` is expected.

Inheritance The definition of one kind of object can be reused to produce a new kind of object. This new definition can override some behavior, but also share code with its parent.

Open recursion An object's methods can invoke another method in the same object using a special variable (often called `self` or `this`). When objects are created from classes, these calls use dynamic lookup, allowing a method defined in one class to invoke methods defined in another class that inherits from the first.

Almost every notable modern programming language has been influenced by OOP,

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

and you'll have run across these terms if you've ever used C++, Java, C#, Ruby, Python, or JavaScript.

13.1 OCaml Objects

If you already know about object-oriented programming in a language like Java or C++, the OCaml object system may come as a surprise. Foremost is the complete separation of objects and their types from the class system. In a language like Java, a class name is also used as the type of objects created by instantiating it, and the relationships between these object types correspond to inheritance. For example, if we implement a class `Deque` in Java by inheriting from a class `Stack`, we would be allowed to pass a deque anywhere a stack is expected.

OCaml is entirely different. Classes are used to construct objects and support inheritance, but classes are not types. Instead, objects have *object types*, and if you want to use objects, you aren't required to use classes at all. Here's an example of a simple object:

```
# open Base;;
# let s = object
    val mutable v = [0; 2]

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
val s : < pop : int option; push : int -> unit > = <obj>
```

The object has an integer list value `v`, a method `pop` that returns the head of `v`, and a method `push` that adds an integer to the head of `v`.

The object type is enclosed in angle brackets `< ... >`, containing just the types of the methods. Fields, like `v`, are not part of the public interface of an object. All interaction with an object is through its methods. The syntax for a method invocation uses the `#` character:

```
# s#pop;;
- : int option = Some 0
# s#push 4;;
- : unit = ()
# s#pop;;
- : int option = Some 4
```

Note that unlike functions, methods can have zero parameters, since the method call is routed to a concrete object instance. That's why the `pop` method doesn't have a `unit` argument, as the equivalent functional version would.

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

Objects can also be constructed by functions. If we want to specify the initial value of the object, we can define a function that takes the value and returns an object:

```
# let stack init = object
    val mutable v = init

    method pop =
      match v with
      | hd :: tl ->
        v <- tl;
        Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
val stack : 'a list -> < pop : 'a option; push : 'a -> unit > = <fun>
# let s = stack [3; 2; 1];;
val s : < pop : int option; push : int -> unit > = <obj>
# s#pop;;
- : int option = Some 3
```

Note that the types of the function `stack` and the returned object now use the polymorphic type `'a`. When `stack` is invoked on a concrete value `[3; 2; 1]`, we get the same object type as before, with type `int` for the values on the stack.

13.2 Object Polymorphism

Like polymorphic variants, methods can be used without an explicit type declaration:

```
# let area sq = sq#width * sq#width;;
val area : < width : int; .. > -> int = <fun>
# let minimize sq : unit = sq#resize 1;;
val minimize : < resize : int -> unit; .. > -> unit = <fun>
# let limit sq = if (area sq) > 100 then minimize sq;;
val limit : < resize : int -> unit; width : int; .. > -> unit = <fun>
```

As you can see, object types are inferred automatically from the methods that are invoked on them.

The type system will complain if it sees incompatible uses of the same method:

```
# let toggle sq b : unit =
    if b then sq#resize `Fullscreen else minimize sq;;
Line 2, characters 51-53:
Error: This expression has type < resize : [> `Fullscreen ] -> unit;
    .. >
       but an expression was expected of type < resize : int -> unit;
    .. >
       Types for method resize are incompatible
```

The `..` in the inferred object types are ellipses, standing for other unspecified methods that the object may have. The type `< width : float; .. >` specifies an object that must have at least a `width` method, and possibly some others as well. Such object types are said to be *open*.

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

We can manually *close* an object type using a type annotation:

```
# let area_closed (sq: < width : int >) = sq#width * sq#width;;
val area_closed : < width : int > -> int = <fun>
# let sq = object
    method width = 30
    method name = "sq"
  end;;
val sq : < name : string; width : int > = <obj>
# area_closed sq;;
Line 1, characters 13-15:
Error: This expression has type < name : string; width : int >
       but an expression was expected of type < width : int >
       The second object type has no method name
```

Elisions Are Polymorphic

The `..` in an open object type is an elision, standing for "possibly more methods." It may not be apparent from the syntax, but an elided object type is actually polymorphic. For example, if we try to write a type definition, we get an "unbound type variable" error:

```
# type square = < width : int; ..>;;
Line 1, characters 1-33:
Error: A type variable is unbound in this type declaration.
       In type < width : Base.int; .. > as 'a the variable 'a is
    unbound
```

This is because `..` is really a special kind of type variable called a *row variable*.

This kind of typing scheme using row variables is called *row polymorphism*. Row polymorphism is also used in polymorphic variant types, and there is a close relationship between objects and polymorphic variants: objects are to records what polymorphic variants are to ordinary variants.

An object of type `< pop : int option; .. >` can be any object with a method `pop : int option`; it doesn't matter how it is implemented. When the method `#pop` is invoked, the actual method that is run is determined by the object. Consider the following function.

```
# let print_pop st = Option.iter ~f:(Stdio.printf "Popped: %d\n")
    st#pop;;
val print_pop : < pop : int option; .. > -> unit = <fun>
```

We can run it on the stack type we defined above, which is based on linked lists.

```
# print_pop (stack [5;4;3;2;1]);;
Popped: 5
- : unit = ()
```

But we could also create a totally different implementation of stacks, using Base's array-based `Stack` module.

```
# let array_stack l = object
    val stack = Stack.of_list l
    method pop = Stack.pop stack
```

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
  end;;
val array_stack : 'a list -> < pop : 'a option > = <fun>
```

And `print_pop` will work just as well on this kind of stack object, despite having a completely different implementation.

```
# print_pop (array_stack [5;4;3;2;1]);;
Popped: 5
- : unit = ()
```

13.3 Immutable Objects

Many people consider object-oriented programming to be intrinsically imperative, where an object is like a state machine. Sending a message to an object causes it to change state, possibly sending messages to other objects.

Indeed, in many programs this makes sense, but it is by no means required. Let's define a function that creates immutable stack objects:

```
# let imm_stack init = object
    val v = init

    method pop =
      match v with
      | hd :: tl -> Some (hd, {< v = tl >})
      | [] -> None

    method push hd =
      {< v = hd :: v >}
  end;;
val imm_stack :
  'a list -> (< pop : ('a * 'b) option; push : 'a -> 'b > as 'b) = <fun>
```

The key parts of this implementation are in the `pop` and `push` methods. The expression `{< ... >}` produces a copy of the current object, with the same type, and the specified fields updated. In other words, the `push hd` method produces a copy of the object, with `v` replaced by `hd :: v`. The original object is not modified:

```
# let s = imm_stack [3; 2; 1];;
val s : < pop : (int * 'a) option; push : int -> 'a > as 'a = <obj>
# let r = s#push 4;;
val r : < pop : (int * 'a) option; push : int -> 'a > as 'a = <obj>
# s#pop;;
- : (int * (< pop : 'a; push : int -> 'b > as 'b)) option as 'a =
Some (3, <obj>)
# r#pop;;
- : (int * (< pop : 'a; push : int -> 'b > as 'b)) option as 'a =
Some (4, <obj>)
```

There are some restrictions on the use of the expression `{< ... >}`. It can be used only within a method body, and only the values of fields may be updated. Method implementations are fixed at the time the object is created; they cannot be changed dynamically.

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

13.4 When to Use Objects

You might wonder when to use objects in OCaml, which has a multitude of alternative mechanisms to express similar concepts. First-class modules are more expressive (a module can include types, while classes and objects cannot). Modules, functors, and data types also offer a wide range of ways to express program structure. In fact, many seasoned OCaml programmers rarely use classes and objects, if at all.

Objects have some advantages over records: they don't require type definitions, and their support for row polymorphism makes them more flexible. However, the heavy syntax and additional runtime cost means that objects are rarely used in place of records.

The real benefits of objects come from the class system. Classes support inheritance and open recursion. Open recursion allows interdependent parts of an object to be defined separately. This works because calls between the methods of an object are determined when the object is instantiated, a form of *late* binding. This makes it possible (and necessary) for one method to refer to other methods in the object without knowing statically how they will be implemented.

In contrast, modules use early binding. If you want to parameterize your module code so that some part of it can be implemented later, you would write a function or functor. This is more explicit, but often more verbose than overriding a method in a class.

In general, a rule of thumb is: use classes and objects in situations where open recursion is a big win. Two good examples are Xavier Leroy's Cryptokit[1], which provides a variety of cryptographic primitives that can be combined in building-block style; and the Camlimages[2] library, which manipulates various graphical file formats. Camlimages also provides a module-based version of the same library, letting you choose between functional and object-oriented styles depending on your problem domain.

We'll introduce you to classes, and examples using open recursion, in Chapter 14 (Classes).

13.5 Subtyping

Subtyping is a central concept in object-oriented programming. It governs when an object with one type *A* can be used in an expression that expects an object of another type *B*. When this is true, we say that *A* is a *subtype* of *B*. More concretely, subtyping restricts when the coercion operator `e :> t` can be applied. This coercion works only if the type of `e` is a subtype of `t`.

[1] http://gallium.inria.fr/~xleroy/software.html#cryptokit
[2] http://cristal.inria.fr/camlimages/

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

13.5.1 Width Subtyping

To explore this, let's define some simple object types for geometric shapes. The generic type `shape` just has a method to compute the area.

```
# type shape = < area : float >;;
type shape = < area : float >
```

Now let's add a type representing a specific kind of shape, as well as a function for creating objects of that type.

```
# type square = < area : float; width : int >;;
type square = < area : float; width : int >
# let square w = object
    method area = Float.of_int (w * w)
    method width = w
  end;;
val square : int -> < area : float; width : int > = <fun>
```

A `square` has a method `area` just like a `shape`, and an additional method `width`. Still, we expect a `square` to be a `shape`, and it is. Note, however, that the coercion `:>` must be explicit:

```
# (square 10 : shape);;
Line 1, characters 2-11:
Error: This expression has type < area : float; width : int >
       but an expression was expected of type shape
       The second object type has no method width
# (square 10 :> shape);;
- : shape = <obj>
```

This form of object subtyping is called *width* subtyping. Width subtyping means that an object type A is a subtype of B, if A has all of the methods of B, and possibly more. A `square` is a subtype of `shape` because it implements all of the methods of `shape`, which in this case means the `area` method.

13.5.2 Depth Subtyping

We can also use *depth* subtyping with objects. Depth subtyping allows us to coerce an object if its individual methods could safely be coerced. So an object type `< m: t1 >` is a subtype of `< m: t2 >` if `t1` is a subtype of `t2`.

First, let's add a new shape type, `circle`:

```
# type circle = < area : float; radius : int >;;
type circle = < area : float; radius : int >
# let circle r = object
    method area = 3.14 *. (Float.of_int r) **. 2.0
    method radius = r
  end;;
val circle : int -> < area : float; radius : int > = <fun>
```

Using that, let's create a couple of objects that each have a `shape` method, one returning a shape of type `circle`:

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
# let coin = object
    method shape = circle 5
    method color = "silver"
  end;;
val coin : < color : string; shape : < area : float; radius : int > >
    = <obj>
```

And the other returning a shape of type `square`:

```
# let map = object
    method shape = square 10
  end;;
val map : < shape : < area : float; width : int > > = <obj>
```

Both these objects have a `shape` method whose type is a subtype of the `shape` type, so they can both be coerced into the object type `< shape : shape >`:

```
# type item = < shape : shape >;;
type item = < shape : shape >
# let items = [ (coin :> item) ; (map :> item) ];;
val items : item list = [<obj>; <obj>]
```

Polymorphic Variant Subtyping

Subtyping can also be used to coerce a polymorphic variant into a larger polymorphic variant type. A polymorphic variant type A is a subtype of B, if the tags of A are a subset of the tags of B:

```
# type num = [ `Int of int | `Float of float ];;
type num = [ `Float of float | `Int of int ]
# type const = [ num | `String of string ];;
type const = [ `Float of float | `Int of int | `String of string ]
# let n : num = `Int 3;;
val n : num = `Int 3
# let c : const = (n :> const);;
val c : const = `Int 3
```

13.5.3 Variance

What about types built from object types? If a `square` is a `shape`, we expect a `square list` to be a `shape list`. OCaml does indeed allow such coercions:

```
# let squares: square list = [ square 10; square 20 ];;
val squares : square list = [<obj>; <obj>]
# let shapes: shape list = (squares :> shape list);;
val shapes : shape list = [<obj>; <obj>]
```

Note that this relies on lists being immutable. It would not be safe to treat a `square array` as a `shape array` because it would allow you to store non-square shapes into what should be an array of squares. OCaml recognizes this and does not allow the coercion:

```
# let square_array: square array = [| square 10; square 20 |];;
val square_array : square array = [|<obj>; <obj>|]
```

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
# let shape_array: shape array = (square_array :> shape array);;
Line 1, characters 32-61:
Error: Type square array is not a subtype of shape array
       The second object type has no method width
```

We say that `'a list` is *covariant* (in `'a`), while `'a array` is *invariant*.

Subtyping function types requires a third class of variance. A function with type `square -> string` cannot be used with type `shape -> string` because it expects its argument to be a `square` and would not know what to do with a `circle`. However, a function with type `shape -> string` can safely be used with type `square -> string`:

```
# let shape_to_string: shape -> string =
    fun s -> Printf.sprintf "Shape(%F)" s#area;;
val shape_to_string : shape -> string = <fun>
# let square_to_string: square -> string =
    (shape_to_string :> square -> string);;
val square_to_string : square -> string = <fun>
```

We say that `'a -> string` is *contravariant* in `'a`. In general, function types are contravariant in their arguments and covariant in their results.

Variance Annotations

OCaml works out the variance of a type using that type's definition. Consider the following simple immutable `Either` type.

```
# module Either = struct
    type ('a, 'b) t =
      | Left of 'a
      | Right of 'b
    let left x = Left x
    let right x = Right x
  end;;
module Either :
  sig
    type ('a, 'b) t = Left of 'a | Right of 'b
    val left : 'a -> ('a, 'b) t
    val right : 'a -> ('b, 'a) t
  end
```

By looking at what coercions are allowed, we can see that the type parameters of the immutable `Either` type are covariant.

```
# let left_square = Either.left (square 40);;
val left_square : (< area : float; width : int >, 'a) Either.t =
  Either.Left <obj>
# (left_square :> (shape,_) Either.t);;
- : (shape, 'a) Either.t = Either.Left <obj>
```

The story is different, however, if the definition is hidden by a signature.

```
# module Abs_either : sig
    type ('a, 'b) t
    val left: 'a -> ('a, 'b) t
    val right: 'b -> ('a, 'b) t
  end = Either;;
module Abs_either :
```

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
  sig
    type ('a, 'b) t
    val left : 'a -> ('a, 'b) t
    val right : 'b -> ('a, 'b) t
  end
```

In this case, OCaml is forced to assume that the type is invariant.

```
# (Abs_either.left (square 40) :> (shape, _) Abs_either.t);;
Line 1, characters 2-29:
Error: This expression cannot be coerced to type (shape, 'b)
    Abs_either.t;
       it has type (< area : float; width : int >, 'a) Abs_either.t
       but is here used with type (shape, 'b) Abs_either.t
       Type < area : float; width : int > is not compatible with type
         shape = < area : float >
       The second object type has no method width
```

We can fix this by adding *variance annotations* to the type's parameters in the signature: + for covariance or - for contravariance:

```
# module Var_either : sig
    type (+'a, +'b) t
    val left: 'a -> ('a, 'b) t
    val right: 'b -> ('a, 'b) t
  end = Either;;
module Var_either :
  sig
    type (+'a, +'b) t
    val left : 'a -> ('a, 'b) t
    val right : 'b -> ('a, 'b) t
  end
```

As you can see, this now allows the coercion once again.

```
# (Var_either.left (square 40) :> (shape, _) Var_either.t);;
- : (shape, 'a) Var_either.t = <abstr>
```

For a more concrete example of variance, let's create some stacks containing shapes by applying our `stack` function to some squares and some circles:

```
# type 'a stack = < pop: 'a option; push: 'a -> unit >;;
type 'a stack = < pop : 'a option; push : 'a -> unit >
# let square_stack: square stack = stack [square 30; square 10];;
val square_stack : square stack = <obj>
# let circle_stack: circle stack = stack [circle 20; circle 40];;
val circle_stack : circle stack = <obj>
```

If we wanted to write a function that took a list of such stacks and found the total area of their shapes, we might try:

```
# let total_area (shape_stacks: shape stack list) =
    let stack_area acc st =
      let rec loop acc =
        match st#pop with
        | Some s -> loop (acc +. s#area)
        | None -> acc
      in
```

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
      loop acc
    in
  List.fold ~init:0.0 ~f:stack_area shape_stacks;;
val total_area : shape stack list -> float = <fun>
```

However, when we try to apply this function to our objects, we get an error:

```
# total_area [(square_stack :> shape stack); (circle_stack :> shape
    stack)];;
Line 1, characters 13-42:
Error: Type square stack = < pop : square option; push : square ->
    unit >
       is not a subtype of
         shape stack = < pop : shape option; push : shape -> unit >
       Type shape = < area : float > is not a subtype of
         square = < area : float; width : int >
       The first object type has no method width
```

As you can see, `square stack` and `circle stack` are not subtypes of `shape stack`. The problem is with the `push` method. For `shape stack`, the `push` method takes an arbitrary `shape`. So if we could coerce a `square stack` to a `shape stack`, then it would be possible to push an arbitrary shape onto `square stack`, which would be an error.

Another way of looking at this is that `< push: 'a -> unit; .. >` is contravariant in `'a`, so `< push: square -> unit; pop: square option >` cannot be a subtype of `< push: shape -> unit; pop: shape option >`.

Still, the `total_area` function should be fine, in principle. It doesn't call `push`, so it isn't making that error. To make it work, we need to use a more precise type that indicates we are not going to be using the `push` method. We define a type `readonly_stack` and confirm that we can coerce the list of `stacks` to it:

```
# type 'a readonly_stack = < pop : 'a option >;;
type 'a readonly_stack = < pop : 'a option >
# let total_area (shape_stacks: shape readonly_stack list) =
    let stack_area acc st =
      let rec loop acc =
        match st#pop with
        | Some s -> loop (acc +. s#area)
        | None -> acc
      in
      loop acc
    in
  List.fold ~init:0.0 ~f:stack_area shape_stacks;;
val total_area : shape readonly_stack list -> float = <fun>
# total_area [(square_stack :> shape readonly_stack); (circle_stack :>
  shape readonly_stack)];;
- : float = 7280.
```

Aspects of this section may seem fairly complicated, but it should be pointed out that this typing *works*, and in the end, the type annotations are fairly minor. In most typed object-oriented languages, these coercions would simply not be possible. For example, in C++, a STL type `list<T>` is invariant in `T`, so it is simply not possible to use `list<square>` where `list<shape>` is expected (at least safely). The situation is similar in Java, although Java has an escape hatch that allows the program to fall

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

back to dynamic typing. The situation in OCaml is much better: it works, it is statically checked, and the annotations are pretty simple.

13.5.4 Narrowing

Narrowing, also called *down casting*, is the ability to coerce an object to one of its subtypes. For example, if we have a list of shapes `shape list`, we might know (for some reason) what the actual type of each shape is. Perhaps we know that all objects in the list have type `square`. In this case, *narrowing* would allow the recasting of the object from type `shape` to type `square`. Many languages support narrowing through dynamic type checking. For example, in Java, a coercion `(Square) x` is allowed if the value `x` has type `Square` or one of its subtypes; otherwise the coercion throws an exception.

Narrowing is *not permitted* in OCaml. Period.

Why? There are two reasonable explanations, one based on a design principle, and another technical (the technical reason is simple: it is hard to implement).

The design argument is this: narrowing violates abstraction. In fact, with a structural typing system like in OCaml, narrowing would essentially provide the ability to enumerate the methods in an object. To check whether an object `obj` has some method `foo : int`, one would attempt a coercion `(obj :> < foo : int >)`.

More pragmatically, narrowing leads to poor object-oriented style. Consider the following Java code, which returns the name of a shape object:

```
String GetShapeName(Shape s) {
  if (s instanceof Square) {
    return "Square";
  } else if (s instanceof Circle) {
    return "Circle";
  } else {
    return "Other";
  }
}
```

Most programmers would consider this code to be awkward, at the least. Instead of performing a case analysis on the type of object, it would be better to define a method to return the name of the shape. Instead of calling `GetShapeName(s)`, we should call `s.Name()` instead.

However, the situation is not always so obvious. The following code checks whether an array of shapes looks like a barbell, composed of two `Circle` objects separated by a `Line`, where the circles have the same radius:

```
boolean IsBarbell(Shape[] s) {
  return s.length == 3 && (s[0] instanceof Circle) &&
    (s[1] instanceof Line) && (s[2] instanceof Circle) &&
        ((Circle) s[0]).radius() == ((Circle) s[2]).radius();
}
```

In this case, it is much less clear how to augment the `Shape` class to support this kind of pattern analysis. It is also not obvious that object-oriented programming is well-suited for this situation. Pattern matching seems like a better fit:

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
# type shape = Circle of { radius : int } | Line of { length: int };;
type shape = Circle of { radius : int; } | Line of { length : int; }
# let is_barbell = function
  | [Circle {radius=r1}; Line _; Circle {radius=r2}] when r1 = r2 ->
    true
  | _ -> false;;
val is_barbell : shape list -> bool = <fun>
```

Regardless, there is a solution if you find yourself in this situation, which is to augment the classes with variants. You can define a method `variant` that injects the actual object into a variant type.

```
# type shape = < variant : repr >
  and circle = < variant : repr; radius : int >
  and line = < variant : repr; length : int >
  and repr =
   | Circle of circle
   | Line of line;;
type shape = < variant : repr >
and circle = < radius : int; variant : repr >
and line = < length : int; variant : repr >
and repr = Circle of circle | Line of line
# let is_barbell = function
  | [s1; s2; s3] ->
     (match s1#variant, s2#variant, s3#variant with
      | Circle c1, Line _, Circle c2 when c1#radius = c2#radius ->
    true
      | _ -> false)
  | _ -> false;;
val is_barbell : < variant : repr; .. > list -> bool = <fun>
```

This pattern works, but it has drawbacks. In particular, the recursive type definition should make it clear that this pattern is essentially equivalent to using variants, and that objects do not provide much value here.

13.5.5 Subtyping Versus Row Polymorphism

There is considerable overlap between subtyping and row polymorphism. Both mechanisms allow you to write functions that can be applied to objects of different types. In these cases, row polymorphism is usually preferred over subtyping because it does not require explicit coercions, and it preserves more type information, allowing functions like the following:

```
# let remove_large l =
  List.filter ~f:(fun s -> Float.(s#area <= 100.)) l;;
val remove_large : (< area : float; .. > as 'a) list -> 'a list = <fun>
```

The return type of this function is built from the open object type of its argument, preserving any additional methods that it may have, as we can see below.

```
# let squares : < area : float; width : int > list =
  [square 5; square 15; square 10];;
val squares : < area : float; width : int > list = [<obj>; <obj>;
    <obj>]
```

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press

```
# remove_large squares;;
- : < area : float; width : int > list = [<obj>; <obj>]
```

Writing a similar function with a closed type and applying it using subtyping does not preserve the methods of the argument: the returned object is only known to have an `area` method:

```
# let remove_large (l: < area : float > list) =
  List.filter ~f:(fun s -> Float.(s#area <= 100.)) l;;
val remove_large : < area : float > list -> < area : float > list =
    <fun>
# remove_large (squares :> < area : float > list );;
- : < area : float > list = [<obj>; <obj>]
```

There are some situations where we cannot use row polymorphism. In particular, row polymorphism cannot be used to place different types of objects in the same container. For example, lists of heterogeneous elements cannot be created using row polymorphism:

```
# let hlist: < area: float; ..> list = [square 10; circle 30];;
Line 1, characters 50-59:
Error: This expression has type < area : float; radius : int >
       but an expression was expected of type < area : float; width :
    int >
       The second object type has no method radius
```

Similarly, we cannot use row polymorphism to store different types of object in the same reference:

```
# let shape_ref: < area: float; ..> ref = ref (square 40);;
val shape_ref : < area : float; width : int > ref =
  {Base.Ref.contents = <obj>}
# shape_ref := circle 20;;
Line 1, characters 14-23:
Error: This expression has type < area : float; radius : int >
       but an expression was expected of type < area : float; width :
    int >
       The second object type has no method radius
```

In both these cases we must use subtyping:

```
# let hlist: shape list = [(square 10 :> shape); (circle 30 :>
    shape)];;
val hlist : shape list = [<obj>; <obj>]
# let shape_ref: shape ref = ref (square 40 :> shape);;
val shape_ref : shape ref = {Base.Ref.contents = <obj>}
# shape_ref := (circle 20 :> shape);;
- : unit = ()
```

https://doi.org/10.1017/9781009129220.015 Published online by Cambridge University Press