11 Functors

Up until now, we've seen OCaml's modules play an important but limited role. In particular, we've used modules to organize code into units with specified interfaces. But OCaml's module system can do much more than that, serving as a powerful tool for building generic code and structuring large-scale systems. Much of that power comes from functors.

Functors are, roughly speaking, functions from modules to modules, and they can be used to solve a variety of code-structuring problems, including:

Dependency injection Makes the implementations of some components of a system swappable. This is particularly useful when you want to mock up parts of your system for testing and simulation purposes.

Autoextension of modules Functors give you a way of extending existing modules with new functionality in a standardized way. For example, you might want to add a slew of comparison operators derived from a base comparison function. To do this by hand would require a lot of repetitive code for each type, but functors let you write this logic just once and apply it to many different types.

Instantiating modules with state Modules can contain mutable states, and that means that you'll occasionally want to have multiple instantiations of a particular module, each with its own separate and independent mutable state. Functors let you automate the construction of such modules.

These are really just some of the uses that you can put functors to. We'll make no attempt to provide examples of all of the uses of functors here. Instead, this chapter will try to provide examples that illuminate the language features and design patterns that you need to master in order to use functors effectively.

11.1 A Trivial Example

Let's create a functor that takes a module containing a single integer variable `x` and returns a new module with `x` incremented by one. This is intended to serve as a way to walk through the basic mechanics of functors, even though it's not something you'd want to do in practice.

First, let's define a signature for a module that contains a single value of type `int`:

```
# open Base;;
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
```

Now we can define our functor. We'll use `X_int` both to constrain the argument to the functor and to constrain the module returned by the functor:

```
# module Increment (M : X_int) : X_int = struct
    let x = M.x + 1
  end;;
module Increment : functor (M : X_int) -> X_int
```

One thing that immediately jumps out is that functors are more syntactically heavyweight than ordinary functions. For one thing, functors require explicit (module) type annotations, which ordinary functions do not. Technically, only the type on the input is mandatory, although in practice, you should usually constrain the module returned by the functor, just as you should use an `mli`, even though it's not mandatory.

The following shows what happens when we omit the module type for the output of the functor:

```
# module Increment (M : X_int) = struct
    let x = M.x + 1
  end;;
module Increment : functor (M : X_int) -> sig val x : int end
```

We can see that the inferred module type of the output is now written out explicitly, rather than being a reference to the named signature `X_int`.

We can use `Increment` to define new modules:

```
# module Three = struct let x = 3 end;;
module Three : sig val x : int end
# module Four = Increment(Three);;
module Four : sig val x : int end
# Four.x - Three.x;;
- : int = 1
```

In this case, we applied `Increment` to a module whose signature is exactly equal to `X_int`. But we can apply `Increment` to any module that *satisfies* the interface `X_int`, in the same way that the contents of an `ml` file must satisfy the `mli`. That means that the module type can omit some information available in the module, either by dropping fields or by leaving some fields abstract. Here's an example:

```
# module Three_and_more = struct
    let x = 3
    let y = "three"
  end;;
module Three_and_more : sig val x : int val y : string end
# module Four = Increment(Three_and_more);;
module Four : sig val x : int end
```

The rules for determining whether a module matches a given signature are similar in spirit to the rules in an object-oriented language that determine whether an object satisfies a given interface. As in an object-oriented context, the extra information that doesn't match the signature you're looking for (in this case, the variable `y`) is simply ignored.

11.2 A Bigger Example: Computing with Intervals

Let's consider a more realistic example of how to use functors: a library for computing with intervals. Intervals are a common computational object, and they come up in different contexts and for different types. You might need to work with intervals of floating-point values or strings or times, and in each of these cases, you want similar operations: testing for emptiness, checking for containment, intersecting intervals, and so on.

We can use functors to build a generic interval library that can be used with any type that supports a total ordering on the underlying set.

First we'll define a module type that captures the information we'll need about the endpoints of the intervals. This interface, which we'll call `Comparable`, contains just two things: a comparison function and the type of the values to be compared:

```
# module type Comparable = sig
    type t
    val compare : t -> t -> int
  end;;
module type Comparable = sig type t val compare : t -> t -> int end
```

The comparison function follows the standard OCaml idiom for such functions, returning `0` if the two elements are equal, a positive number if the first element is larger than the second, and a negative number if the first element is smaller than the second. Thus, we could rewrite the standard comparison functions on top of `compare`.

```
compare x y < 0     (* x < y *)
compare x y = 0     (* x = y *)
compare x y > 0     (* x > y *)
```

(This idiom is a bit of a historical error. It would be better if `compare` returned a variant with three cases for less than, greater than, and equal. But it's a well-established idiom at this point, and unlikely to change.)

The functor for creating the interval module follows. We represent an interval with a variant type, which is either `Empty` or `Interval (x,y)`, where `x` and `y` are the bounds of the interval. In addition to the type, the body of the functor contains implementations of a number of useful primitives for interacting with intervals:

```
# module Make_interval(Endpoint : Comparable) = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false
      (** [contains t x] returns true iff [x] is contained in the
          interval [t] *)
      let contains t x =
        match t with
        | Empty -> false
        | Interval (l,h) ->
          Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

      (** [intersect t1 t2] returns the intersection of the two input
          intervals *)
      let intersect t1 t2 =
        let min x y = if Endpoint.compare x y <= 0 then x else y in
        let max x y = if Endpoint.compare x y >= 0 then x else y in
        match t1,t2 with
        | Empty, _ | _, Empty -> Empty
        | Interval (l1,h1), Interval (l2,h2) ->
          create (max l1 l2) (min h1 h2)

    end;;
  module Make_interval :
    functor (Endpoint : Comparable) ->
      sig
        type t = Interval of Endpoint.t * Endpoint.t | Empty
        val create : Endpoint.t -> Endpoint.t -> t
        val is_empty : t -> bool
        val contains : t -> Endpoint.t -> bool
        val intersect : t -> t -> t
      end
```

We can instantiate the functor by applying it to a module with the right signature. In the following code, rather than name the module first and then call the functor, we provide the functor input as an anonymous module:

```
  # module Int_interval =
      Make_interval(struct
        type t = int
        let compare = Int.compare
    end);;
  module Int_interval :
    sig
      type t = Interval of int * int | Empty
      val create : int -> int -> t
      val is_empty : t -> bool
      val contains : t -> int -> bool
      val intersect : t -> t -> t
    end
```

If the input interface for your functor is aligned with the standards of the libraries you use, then you don't need to construct a custom module to feed to the functor. In this case, we can directly use the `Int` or `String` modules provided by `Base`:

```
  # module Int_interval = Make_interval(Int);;
  module Int_interval :
    sig
      type t = Make_interval(Base.Int).t = Interval of int * int | Empty
      val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
# module String_interval = Make_interval(String);;
module String_interval :
  sig
    type t =
      Make_interval(Base.String).t =
        Interval of string * string
      | Empty
    val create : string -> string -> t
    val is_empty : t -> bool
    val contains : t -> string -> bool
    val intersect : t -> t -> t
  end
```

This works because many modules in Base, including `Int` and `String`, satisfy an extended version of the `Comparable` signature described previously. Such standardized signatures are good practice, both because they make functors easier to use, and because they encourage standardization that makes your codebase easier to navigate.

We can use the newly defined `Int_interval` module like any ordinary module:

```
# let i1 = Int_interval.create 3 8;;
val i1 : Int_interval.t = Int_interval.Interval (3, 8)
# let i2 = Int_interval.create 4 10;;
val i2 : Int_interval.t = Int_interval.Interval (4, 10)
# Int_interval.intersect i1 i2;;
- : Int_interval.t = Int_interval.Interval (4, 8)
```

This design gives us the freedom to use any comparison function we want for comparing the endpoints. We could, for example, create a type of integer interval with the order of the comparison reversed, as follows:

```
# module Rev_int_interval =
    Make_interval(struct
      type t = int
      let compare x y = Int.compare y x
  end);;
module Rev_int_interval :
  sig
    type t = Interval of int * int | Empty
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
```

The behavior of `Rev_int_interval` is of course different from `Int_interval`:

```
# let interval = Int_interval.create 4 3;;
val interval : Int_interval.t = Int_interval.Empty
# let rev_interval = Rev_int_interval.create 4 3;;
val rev_interval : Rev_int_interval.t = Rev_int_interval.Interval (4,
    3)
```

Importantly, `Rev_int_interval.t` is a different type than `Int_interval.t`, even though its physical representation is the same. Indeed, the type system will prevent us from confusing them.

```
# Int_interval.contains rev_interval 3;;
Line 1, characters 23-35:
Error: This expression has type Rev_int_interval.t
       but an expression was expected of type Int_interval.t
```

This is important, because confusing the two kinds of intervals would be a semantic error, and it's an easy one to make. The ability of functors to mint new types is a useful trick that comes up a lot.

11.2.1 Making the Functor Abstract

There's a problem with `Make_interval`. The code we wrote depends on the invariant that the upper bound of an interval is greater than its lower bound, but that invariant can be violated. The invariant is enforced by the `create` function, but because `Int_interval.t` is not abstract, we can bypass the `create` function:

```
# Int_interval.is_empty (* going through create *)
  (Int_interval.create 4 3);;
- : bool = true
# Int_interval.is_empty (* bypassing create *)
  (Int_interval.Interval (4,3));;
- : bool = false
```

To make `Int_interval.t` abstract, we need to restrict the output of `Make_interval` with an interface. Here's an explicit interface that we can use for that purpose:

```
# module type Interval_intf = sig
    type t
    type endpoint
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end;;
module type Interval_intf =
  sig
    type t
    type endpoint
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end
```

This interface includes the type `endpoint` to give us a way of referring to the endpoint type. Given this interface, we can redo our definition of `Make_interval`. Notice that we added the type `endpoint` to the implementation of the module to match `Interval_intf`:

```
# module Make_interval(Endpoint : Comparable) : Interval_intf = struct
    type endpoint = Endpoint.t
    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty
    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    (** [contains t x] returns true iff [x] is contained in the
        interval [t] *)
    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    (** [intersect t1 t2] returns the intersection of the two input
        intervals *)
    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
        create (max l1 l2) (min h1 h2)

  end;;
module Make_interval : functor (Endpoint : Comparable) ->
    Interval_intf
```

11.2.2 Sharing Constraints

The resulting module is abstract, but it’s unfortunately too abstract. In particular, we haven’t exposed the type endpoint, which means that we can’t even construct an interval anymore:

```
# module Int_interval = Make_interval(Int);;
module Int_interval :
  sig
    type t = Make_interval(Base.Int).t
    type endpoint = Make_interval(Base.Int).endpoint
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end
# Int_interval.create 3 4;;
Line 1, characters 21-22:
Error: This expression has type int but an expression was expected of
    type
        Int_interval.endpoint
```

To fix this, we need to expose the fact that `endpoint` is equal to `Int.t` (or more generally, `Endpoint.t`, where `Endpoint` is the argument to the functor). One way of doing this is through a *sharing constraint*, which allows you to tell the compiler to expose the fact that a given type is equal to some other type. The syntax for a simple sharing constraint is as follows:

```
<Module_type> with type <type> = <type'>
```

The result of this expression is a new signature that's been modified so that it exposes the fact that *`type`* defined inside of the module type is equal to *`type'`* whose definition is outside of it. One can also apply multiple sharing constraints to the same signature:

```
<Module_type> with type <type1> = <type1'> and type <type2> = <type2'>
```

We can use a sharing constraint to create a specialized version of `Interval_intf` for integer intervals:

```
# module type Int_interval_intf =
  Interval_intf with type endpoint = int;;
module type Int_interval_intf =
  sig
    type t
    type endpoint = int
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end
```

We can also use sharing constraints in the context of a functor. The most common use case is where you want to expose that some of the types of the module being generated by the functor are related to the types in the module fed to the functor.

In this case, we'd like to expose an equality between the type `endpoint` in the new module and the type `Endpoint.t`, from the module `Endpoint` that is the functor argument. We can do this as follows:

```
# module Make_interval(Endpoint : Comparable)
    : (Interval_intf with type endpoint = Endpoint.t)
  = struct

    type endpoint = Endpoint.t
    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false
    (** [contains t x] returns true iff [x] is contained in the
        interval [t] *)
    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    (** [intersect t1 t2] returns the intersection of the two input
        intervals *)
    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
        create (max l1 l2) (min h1 h2)

  end;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t
      type endpoint = Endpoint.t
      val create : endpoint -> endpoint -> t
      val is_empty : t -> bool
      val contains : t -> endpoint -> bool
      val intersect : t -> t -> t
    end
```

Now the interface is as it was, except that `endpoint` is known to be equal to `Endpoint.t`. As a result of that type equality, we can again do things that require that `endpoint` be exposed, like constructing intervals:

```
# module Int_interval = Make_interval(Int);;
module Int_interval :
  sig
    type t = Make_interval(Base.Int).t
    type endpoint = int
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end
# let i = Int_interval.create 3 4;;
val i : Int_interval.t = <abstr>
# Int_interval.contains i 5;;
- : bool = false
```

11.2.3 Destructive Substitution

Sharing constraints basically do the job, but they have some downsides. In particular, we've now been stuck with the useless type declaration of `endpoint` that clutters up both the interface and the implementation. A better solution would be to modify the

Interval_intf signature by replacing endpoint with Endpoint.t everywhere it shows up, and deleting the definition of endpoint from the signature. We can do just this using what's called *destructive substitution*. Here's the basic syntax:

```
<Module_type> with type <type> := <type'>
```

This looks just like a sharing constraint, except that we use := instead of =. The following shows how we could use this with Make_interval.

```
# module type Int_interval_intf =
  Interval_intf with type endpoint := int;;
module type Int_interval_intf =
  sig
    type t
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
```

There's now no endpoint type: all of its occurrences have been replaced by int. As with sharing constraints, we can also use this in the context of a functor:

```
# module Make_interval(Endpoint : Comparable)
    : Interval_intf with type endpoint := Endpoint.t =
  struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    (** [contains t x] returns true iff [x] is contained in the
        interval [t] *)
    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    (** [intersect t1 t2] returns the intersection of the two input
        intervals *)
    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
```

```
        create (max l1 l2) (min h1 h2)

  end;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
      val contains : t -> Endpoint.t -> bool
      val intersect : t -> t -> t
    end
```

The interface is precisely what we want: the type `t` is abstract, and the type of the endpoint is exposed; so we can create values of type `Int_interval.t` using the creation function, but not directly using the constructors and thereby violating the invariants of the module.

```
# module Int_interval = Make_interval(Int);;
module Int_interval :
  sig
    type t = Make_interval(Base.Int).t
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
# Int_interval.is_empty
  (Int_interval.create 3 4);;
- : bool = false
# Int_interval.is_empty (Int_interval.Interval (4,3));;
Line 1, characters 24-45:
Error: Unbound constructor Int_interval.Interval
```

In addition, the `endpoint` type is gone from the interface, meaning we no longer need to define the `endpoint` type alias in the body of the module.

It's worth noting that the name is somewhat misleading, in that there's nothing destructive about destructive substitution; it's really just a way of creating a new signature by transforming an existing one.

11.2.4 Using Multiple Interfaces

Another feature that we might want for our interval module is the ability to *serialize*, i.e., to be able to read and write intervals as a stream of bytes. In this case, we'll do this by converting to and from s-expressions, which were mentioned already in Chapter 8 (Error Handling). To recall, an s-expression is essentially a parenthesized expression whose atoms are strings, and it is a serialization format that is used commonly in `Base`. Here's an example:

```
# Sexp.List [ Sexp.Atom "This"; Sexp.Atom "is"
  ; Sexp.List [Sexp.Atom "an"; Sexp.Atom "s-expression"]];;
- : Sexp.t = (This is (an s-expression))
```

Base is designed to work well with a syntax extension called ppx_sexp_conv which will generate s-expression conversion functions for any type annotated with [@@deriving sexp]. We can enable ppx_sexp_conv along with a collection of other useful extensions by enabling ppx_jane:

```
# #require "ppx_jane";;
```

Now, we can use the deriving annotation to create sexp-converters for a given type.

```
# type some_type = int * string list [@@deriving sexp];;
type some_type = int * string list
val some_type_of_sexp : Sexp.t -> some_type = <fun>
val sexp_of_some_type : some_type -> Sexp.t = <fun>
# sexp_of_some_type (33, ["one"; "two"]);;
- : Sexp.t = (33 (one two))
# Core.Sexp.of_string "(44 (five six))" |> some_type_of_sexp;;
- : some_type = (44, ["five"; "six"])
```

We'll discuss s-expressions and Sexplib in more detail in Chapter 21 (Data Serialization with S-Expressions), but for now, let's see what happens if we attach the [@@deriving sexp] declaration to the definition of t within the functor:

```
# module Make_interval(Endpoint : Comparable)
    : (Interval_intf with type endpoint := Endpoint.t) = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty
    [@@deriving sexp]

    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    (** [contains t x] returns true iff [x] is contained in the
        interval [t] *)
    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    (** [intersect t1 t2] returns the intersection of the two input
        intervals *)
    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
        create (max l1 l2) (min h1 h2)
```

```
  end;;
Line 4, characters 28-38:
Error: Unbound value Endpoint.t_of_sexp
```

The problem is that [@@deriving sexp] adds code for defining the s-expression converters, and that code assumes that Endpoint has the appropriate sexp-conversion functions for Endpoint.t. But all we know about Endpoint is that it satisfies the Comparable interface, which doesn't say anything about s-expressions.

Happily, Base comes with a built-in interface for just this purpose called Sexpable.S, which is defined as follows:

```
sig
  type t
  val sexp_of_t : t -> Sexp.t
  val t_of_sexp : Sexp.t -> t
end
```

We can modify Make_interval to use the Sexpable.S interface, for both its input and its output. First, let's create an extended version of the Interval_intf interface that includes the functions from the Sexpable.S interface. We can do this using destructive substitution on the Sexpable.S interface, to avoid having multiple distinct type t's clashing with each other:

```
# module type Interval_intf_with_sexp = sig
    include Interval_intf
    include Sexpable.S with type t := t
  end;;
module type Interval_intf_with_sexp =
  sig
    type t
    type endpoint
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
    val t_of_sexp : Sexp.t -> t
    val sexp_of_t : t -> Sexp.t
  end
```

Equivalently, we can define a type t within our new module, and apply destructive substitutions to all of the included interfaces, Interval_intf included, as shown in the following example. This is somewhat cleaner when combining multiple interfaces, since it correctly reflects that all of the signatures are being handled equivalently:

```
# module type Interval_intf_with_sexp = sig
    type t
    include Interval_intf with type t := t
    include Sexpable.S with type t := t
  end;;
module type Interval_intf_with_sexp =
  sig
    type t
    type endpoint
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
    val t_of_sexp : Sexp.t -> t
    val sexp_of_t : t -> Sexp.t
  end
```

Now we can write the functor itself. We have been careful to override the sexp converter here to ensure that the data structure's invariants are still maintained when reading in from an s-expression:

```
# module Make_interval(Endpoint : sig
      type t
      include Comparable with type t := t
      include Sexpable.S with type t := t
    end)
    : (Interval_intf_with_sexp with type endpoint := Endpoint.t)
  = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty
    [@@deriving sexp]

    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (* put a wrapper around the autogenerated [t_of_sexp] to enforce
       the invariants of the data structure *)
    let t_of_sexp sexp =
      match t_of_sexp sexp with
      | Empty -> Empty
      | Interval (x,y) -> create x y

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    (** [contains t x] returns true iff [x] is contained in the
        interval [t] *)
    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    (** [intersect t1 t2] returns the intersection of the two input
        intervals *)
    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
```

```
          create (max l1 l2) (min h1 h2)
    end;;
  module Make_interval :
    functor
      (Endpoint : sig
                    type t
                    val compare : t -> t -> int
                    val t_of_sexp : Sexp.t -> t
                    val sexp_of_t : t -> Sexp.t
                  end)
      ->
      sig
        type t
        val create : Endpoint.t -> Endpoint.t -> t
        val is_empty : t -> bool
        val contains : t -> Endpoint.t -> bool
        val intersect : t -> t -> t
        val t_of_sexp : Sexp.t -> t
        val sexp_of_t : t -> Sexp.t
      end
```

Finally, we can use that sexp converter in the ordinary way:

```
# module Int_interval = Make_interval(Int);;
module Int_interval :
  sig
    type t = Make_interval(Base.Int).t
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
    val t_of_sexp : Sexp.t -> t
    val sexp_of_t : t -> Sexp.t
  end
# Int_interval.sexp_of_t (Int_interval.create 3 4);;
- : Sexp.t = (Interval 3 4)
# Int_interval.sexp_of_t (Int_interval.create 4 3);;
- : Sexp.t = Empty
```

11.3 Extending Modules

Another common use of functors is to generate type-specific functionality for a given module in a standardized way. Let's see how this works in the context of a functional queue, which is just a functional version of a FIFO (first-in, first-out) queue. Being functional, operations on the queue return new queues, rather than modifying the queues that were passed in.

Here's a reasonable `mli` for such a module:

```
type 'a t

val empty : 'a t

(** [enqueue q el] adds [el] to the back of [q] *)
```

```
val enqueue : 'a t -> 'a -> 'a t

(** [dequeue q] returns None if the [q] is empty, otherwise returns
    the first element of the queue and the remainder of the queue *)
val dequeue : 'a t -> ('a * 'a t) option

(** Folds over the queue, from front to back *)
val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
```

The signature of the `fold` function requires some explanation. It follows the same pattern as the `List.fold` function we described in Chapter 4.4 (Using the List Module Effectively). Essentially, `Fqueue.fold q ~init ~f` walks over the elements of `q` from front to back, starting with an accumulator of `init` and using `f` to update the accumulator value as it walks over the queue, returning the final value of the accumulator at the end of the computation. `fold` is a quite powerful operation, as we'll see.

We'll implement `Fqueue` using the well known trick of maintaining an input and an output list so that one can both efficiently enqueue on the input list and dequeue from the output list. If you attempt to dequeue when the output list is empty, the input list is reversed and becomes the new output list. Here's the implementation:

```
open Base

type 'a t = 'a list * 'a list

let empty = ([],[])

let enqueue (in_list, out_list) x =
  (x :: in_list,out_list)

let dequeue (in_list, out_list) =
  match out_list with
  | hd :: tl -> Some (hd, (in_list, tl))
  | [] ->
    match List.rev in_list with
    | [] -> None
    | hd :: tl -> Some (hd, ([], tl))

let fold (in_list, out_list) ~init ~f =
  let after_out = List.fold ~init ~f out_list in
  List.fold_right ~init:after_out ~f:(fun x acc -> f acc x) in_list
```

One problem with `Fqueue` is that the interface is quite skeletal. There are lots of useful helper functions that one might want that aren't there. The `List` module, by way of contrast, has functions like `List.iter`, which runs a function on each element; and `List.for_all`, which returns true if and only if the given predicate evaluates to `true` on every element of the list. Such helper functions come up for pretty much every container type, and implementing them over and over is a dull and repetitive affair.

As it happens, many of these helper functions can be derived mechanically from the `fold` function we already implemented. Rather than write all of these helper functions by hand for every new container type, we can instead use a functor to add this functionality to any container that has a `fold` function.

We'll create a new module, `Foldable`, that automates the process of adding helper functions to a fold-supporting container. As you can see, Foldable contains a module signature S which defines the signature that is required to support folding; and a functor Extend that allows one to extend any module that matches Foldable.S:

```
open Base

module type S = sig
  type 'a t
  val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
end

module type Extension = sig
  type 'a t
  val iter    : 'a t -> f:('a -> unit) -> unit
  val length  : 'a t -> int
  val count   : 'a t -> f:('a -> bool) -> int
  val for_all : 'a t -> f:('a -> bool) -> bool
  val exists  : 'a t -> f:('a -> bool) -> bool
end

(* For extending a Foldable module *)
module Extend(Arg : S)
  : (Extension with type 'a t := 'a Arg.t) =
struct
  open Arg

  let iter t ~f =
    fold t ~init:() ~f:(fun () a -> f a)

  let length t =
    fold t ~init:0  ~f:(fun acc _ -> acc + 1)

  let count t ~f =
    fold t ~init:0  ~f:(fun count x -> count + if f x then 1 else 0)

  exception Short_circuit

  let for_all c ~f =
    try iter c ~f:(fun x -> if not (f x) then raise Short_circuit);
    true
    with Short_circuit -> false

  let exists c ~f =
    try iter c ~f:(fun x -> if f x then raise Short_circuit); false
    with Short_circuit -> true
end
```

Now we can apply this to Fqueue. We can create an interface for an extended version of Fqueue as follows:

```
type 'a t
include (module type of Fqueue) with type 'a t := 'a t
include Foldable.Extension with type 'a t := 'a t
```

In order to apply the functor, we'll put the definition of Fqueue in a submodule called T, and then call Foldable.Extend on T:

```
include Fqueue
include Foldable.Extend(Fqueue)
```

`Base` comes with a number of functors for extending modules that follow this same basic pattern, including:

- `Container.Make` : Very similar to `Foldable.Extend`.
- `Comparable.Make` : Adds support for functionality that depends on the presence of a comparison function, including support for containers like maps and sets.
- `Hashable.Make` : Adds support for hashing-based data structures including hash tables, hash sets, and hash heaps.
- `Monad.Make` : For so-called monadic libraries, like those discussed in Chapters Chapter 8 (Error Handling) and Chapter 17 (Concurrent Programming with Async). Here, the functor is used to provide a collection of standard helper functions based on the `bind` and `return` operators.

These functors come in handy when you want to add the same kind of functionality that is commonly available in `Base` to your own types.

We've really only covered some of the possible uses of functors. Functors are really a quite powerful tool for modularizing your code. The cost is that functors are syntactically heavyweight compared to the rest of the language, and that there are some tricky issues you need to understand to use them effectively, with sharing constraints and destructive substitution being high on that list.

All of this means that for small and simple programs, heavy use of functors is probably a mistake. But as your programs get more complicated and you need more effective modular architectures, functors become a highly valuable tool.