1 Overview

Programs, especially large ones, are typically decomposed into components that can be separately written, compiled, tested, and debugged. Ocaml and many other languages call such components modules, but they are also known as packages, structures, units, and classes.\(^1\)

Ideally, each individual module is described by an interface that specifies the components required by the module from the rest of the program (the imports) and the components supplied by the module to the rest of the program (the exports). Interfaces often list the names and types of imported and exported values along with informal English descriptions of these values (e.g., Java APIs). Such interfaces make it possible for programmers to implement a module without having to know the implementation details of other modules. They also make it possible for a compiler to check for type consistency within a single module.

The process of combining modules to form a whole program is called linking. Linking is typically performed in a distinct link time phase that is performed after all the individual modules are compiled (compile time) but before the entire program is executed (run time).

The specification for how to combine the modules to form a program is written in a linking language. The linking language is almost always different from the programming language in which the module components are written. A crude form of linking involves hard-wiring the file names for imported modules within the source code for a given module. In more flexible approaches, a module is parameterized over names for the imported modules and the linking language specifies the actual modules to be used for the parameters. Ideally, the linking language should check that the interface types of the actual module parameters are consistent with those of the formal module parameters. In this case, the linking language is effectively a simple typed programming language.

Often, a linking language simply lists the modules to be combined. For example, the object files of a C program are linked by supplying a list of file names to the compiler (which actually calls the linker). A linking language can be made more powerful by adding other programming language features that allow more computation to be performed during the linking process. But the desire to make linking languages more expressive is often in tension with the desire to guarantee that (1) the linking process terminates and (2) mere mortals can reliably understand and use the sophisticated types that often accompany more expressive linking languages.

Modules typically serve several purposes in a programming language:

- **Modular Program Structure:** As noted above, modules are used to decompose big programs into smaller parts that can be separately written, compiled, tested, and debugged. This facilitates dividing a program project among members of a programming team. Each team member can work on his or her part independently, and the parts can later be linked to form a working program. Modules also help individual programmers to organize their work by allowing them to concentrate on one part of a problem at a time.

- **Name Control:** Modules help to control the use of names in a program. It is often natural to use the same name for different values in different parts of a program. For example, the name map may mean a list-mapping function in one part of a program and a tree-mapping function in another part of a program. Qualifying the name map with the name of its module,

\(^1\)In many languages, such as C, files serve as de facto modules, but in general the relationship between source files and program modules can be more complex.
as in List.map and Tree.map, allows the programmer to indicate which map is meant in a particular context. Modules typically provide a way to distinguish which values should be visible to the rest of the program (public in JAVA) and which values should remain hidden within the module (private in JAVA).

- **Data Abstraction:** In many languages (including OCAML), modules are the means of separating the specification of a data abstraction from its implementation. Ideally, multiple implementations of the same data abstraction should be allowed to co-exist within a single program.

In the rest of this handout, we explore modules using the OCAML module system as an example. Any module system will have a means of specification (the interface), implementation, and combination. In OCAML, a module specification is called a **signature**, and a module implementation is called a **structure**. Modules can be combined using direct references in the form of fully qualified names (or via open), but more sophisticated operations of abstracting and specializing modules are available via **functors**. We shall explore each of these facilities below.

## 2 Structures

In OCAML, we can collect declarations into a module using the notation:

```ocaml
struct module-declarations end
```

This creates a an entity called a **structure**, which is OCAML’s terminology for a module implementation. A structure can be named using the notation:

```ocaml
module module-name = structure
```

For example, Fig. 1 shows a structure named **Circ** that contains useful values for dealing with circles.

```ocaml
module Circ = struct
  let pi = 3.14159
  let circum r = 2.0 *. pi *. r
  let sq x = x *. x
  let area r = pi *. (sq r)
end
```

Figure 1: The Circ module.

OCAML uses **qualified names** of the form *module-name.component-name* (“dot notation”) to extract module components from a module. The following code shows how to refer to Circ components from outside the Circ module:

```
# Circ.pi *. 10.0;;
- : float = 31.4159
# (Circ.circum 10.0, Circ.area 10.0);
- : float * float = (62.8318, 314.159)
```

Qualified names are important for distinguishing values that have the same component name in two different modules. For example, suppose there is a Rect module containing the following area declaration:

```
let area w h = w *. h
```

Then we can use Circ.area and Rect.area in the same expression:
Using qualified names everywhere can be cumbersome. The OCAML open declaration “opens up” a module and permits its components to be used with their unqualified names. For example, the declaration open Circ is equivalent to the following sequence of declarations:

\[
\begin{align*}
\text{let } &\pi = \text{Circ.pi}; \\
\text{let } &\text{circum} = \text{Circ.circum}; \\
\text{let } &\text{sq} = \text{Circ.sq}; \\
\text{let } &\text{area} = \text{Circ.area};
\end{align*}
\]

The open declaration can be used in the top-level OCAML interpreter or inside a structure. For example, here is sample top-level use:

```
# open Circ;;
# (circum 10.0, area 10.0);;
- : float * float = (62.8318, 314.159)
```

As an example of using open within a structure, consider:

```ocaml
module TestCirc = struct
  open Circ
  let test1 r = (circum r, area r)
  let test2 r = (sq pi +. circum r +. area r)
end
```

In this case, the open declaration permits the use of unqualified names from the Circ module in the remainder of the TestCirc declarations.

It is possible to open multiple modules within a structure declaration. If two modules export the same name, the unqualified name refers to the component from the module opened last. For example:

```ocaml
module Test2 = struct
  open Circ
  let f r = (circum r, area r)
  open Rect
  let g x y = (circum x, Circ.area y, area x y)
end
```

The unqualified area in f refers to Circ.area. However, the unqualified area in g refers to Rect.area, since Rect was most recently opened. Using Circ.area within g requires explicit qualification to distinguish it from Rect.area.

The module declaration can be used to introduce synonyms for structure names within another structure. In the following module, the Circ and Rect modules are not opened but are given one-letter abbreviations that makes the explicitly qualified names more concise.

```ocaml
module Test3 = struct
  module C = Circ
  module R = Rect
  let f r = (C.circum r, C.area r)
  let g x y = (C.circum x, C.area y, R.area x y)
end
```

We may also use one module declaration within another to define nested structures. An example of this is shown in Fig. 2. A sequence of module qualifications can be used to extract the innermost components:
OCAML structures are somewhat like records/structs/objects in other languages. For example, dot notation is used to extract record components in PASCAL, struct components in C, and object components in JAVA. There are two key differences between OCAML structures and traditional record values:

1. OCAML structures can include type components as well as value components. We shall encounter this feature when we study abstract data types in Sec. 4. Handling modules with type components requires a sophisticated type system.

2. Unlike traditional record values, structures are second-class entities in OCAML — they can be manipulated only in limited ways. For instance, structures cannot be named with a let, passed as arguments to functions, returned from functions as results, or stored in data structures. This limitation is imposed to simplify the type system and the linking process.

The second-classness of OCAML structures makes them like JAVA classes, which are also second-class entities. Indeed, the examples we have seen so far (except nested modules) can be expressed in JAVA using classes containing static variables and methods. For example, here is the Circ module expressed in JAVA:

```java
public class Circ {
    public static double pi = 3.14159;
    public static double circum (double r) {return 2*pi*r;}
    public static double sq (double x) {return x*x;}
    public static double area (double r) {return pi*sq(r);}
}
```

As in OCAML, qualified names are used used in JAVA to extract a static component from a class (e.g., Circ.circum(10.0)).

JAVA classes are a form of module, but JAVA has another module mechanism, the package, for collecting related classes together. JAVA’s import declaration is similar to OCAML’s open declaration, but it works at the package level rather than at the class level. For example, in a regular JAVA program, a two-dimensional point can be created as follows:

```java
2OCAML also provides traditional record structures that are first class.
```
new java.awt.Point(1,2)

The qualified name `java.awt.Point` indicates that the `Point` class can be found in the `java.awt` package. However, if the declaration

```java
import java.awt.Point;
```

appears at the top of the file, the class name `Point` may be used without qualification.

## 3 Signatures

If the structure in Fig. 1 is stored in the file `Circ.ml`, then we can load it into the top-level interpreter as follows:

```ocaml
# #use "Circ.ml";;
module Circ :
  sig
    val pi : float
    val circum : float -> float
    val sq : float -> float
    val area : float -> float
  end
```

A module has a type, which is called its **signature**. A signature consists of a collection of declaration types between keywords `sig` and `end`. A signature may even include nested `module` declarations, as exemplified by the `Nested` module:

```ocaml
# #use "Nested.ml";;
module Nested :
  sig
    module Data :
      sig
        val d1 : float list
        val d2 : float list
      end
    module Funs :
      sig
        val f1 : float list -> float list
        val f2 : float list -> float list
      end
  end
```

The declarations of modules opened within a module do not appear in the signature of the module. For example, `TestCirc` opens the `Circ` module, but there is no indication of this in its signature:

```ocaml
# #use "TestCirc.ml";;
module TestCirc :
  sig
    val test1 : float -> float * float
    val test2 : float -> float
  end
```

However, when the `module` declaration is used to rename a module within another module (as in the `Test3` example above), the declarations of the renamed modules appear in the signature:
# use "Test3.ml";
module Test3 :
sig
 module C :
   sig
     val pi : float
     val circum : float -> float
     val sq : float -> float
     val area : float -> float
   end
 module R :
   sig
     val area : float -> float -> float
   end
 val f : float -> float * float
 val g : float -> float -> float * float * float
end

It is possible to name signatures and to declare that structures have a signature. The notation
module type signature-name = signature
introduces a named signature. For instance, here is a signature named CIRC\(^3\) that describes the
values in the Circ module:
module type CIRC = sig
  val pi : float
  val circum : float -> float
  val sq : float -> float
  val area : float -> float
end

We can declare that a structure has a particular signature by writing
module module-name : signature = structure,
where signature is either a signature name, or an explicit signature of the form sig ... end. For
example:
module Circ:CIRC = struct
  let pi = 3.14159
  let circum r = 2.0 *. pi *. r
  let sq x = x *. x
  let area r = pi *. (sq r)
end

Suppose we store the CIRC signature in the file Circ.sig and the modified Circ structure in
the file Circ.ml.\(^4\) Then we can load these into the OCAML interpreter:

\(^3\)Many OCAML programmers name signatures with all caps, but this is only a convention.
\(^4\)We do not have to store the signature and structure in different files. A single file can contain any number of
signatures and modules.
Note how the OCAML interpreter uses the notation module Circ : CIRC to indicate that the Circ structure has the CIRC signature.

Signatures can be used to hide module components. When a module is given an explicit signature, only the names mentioned in the signature are exported from the module; no other names can be extracted from the module. For example, we can defined a restricted version Circres of the Circ module as follows:

```ocaml
# module Circres: sig
  val circum : float -> float
  val area : float -> float
end = Circ;;
module Circres : sig
  val circum : float -> float
  val area : float -> float
end
```

The Circres module exports only the circum and area functions. The other values of the Circ module (pi and sq) are not exported. For example, we cannot use Circres.pi or Circres.sq even though these are used internally to to define Circres.area. In this way, explicit signatures can be used to hide module components that would be declared private in a JAVA class.

## 4 Abstract Data Types

OCAML modules can contain type components as well as value components. In conjunction with this feature, the hiding feature of signatures is ideal for realizing an abstract data type (ADT), in which a contract serves as an abstraction barrier that separates the client and implementer of functions that manipulate an abstract value.

### 4.1 Example: Points

For example, the following signature describes an abstract point type:

```ocaml
module type POINT = sig
  type point
  val make : int -> int -> point
  val getX : point -> int
  val getY : point -> int
  val origin : point
end
```

The declaration type point indicates that any module matching the POINT signature must have a point type, but it does not reveal what the point type is: the point type is abstract.

Here is a structure that implements points as pairs of integers:
# module PairPoint : POINT = struct
  type point = int * int
  let make x y = (x,y)
  let getX (x,_) = x
  let getY (_,y) = y
  let origin = (0,0)
end;;

module PairPoint : POINT

The feedback from the OCaml interpreter, module PairPoint : POINT, indicates that this structure indeed satisfies the POINT signature. We can now manipulate points using values in the PairPoint structure:

# let p = PairPoint.make 1 2;;
val p : PairPoint.point = <abstr>
# PairPoint.getX p;;
- : int = 1
# PairPoint.getY p;;
- : int = 2
# PairPoint.origin;;
- : PairPoint.point = <abstr>
# (PairPoint.getX PairPoint.origin, PairPoint.getY PairPoint.origin);;
- : int * int = (0, 0)

Note how OCaml uses the qualified name PairPoint.point for the type of points created with the PairPoint module. It does not divulge any details about the representation of this type, but uses the notation <abstr> to keep the abstract type hidden. In fact, any attempt to manipulate a point as a pair signals an error:

# fst p;;
Characters 4-5:
  fst p;;

This expression has type PairPoint.point but is here used with type 'a * 'b

Of course, we can implement points using representations other than pairs. For example, we can represent a point as a list of two integers:

# module ListPoint : POINT = struct
  type point = int list
  let make x y = [x;y]
  let getX p = List.hd p
  let getY p = List.hd (List.tl p)
  let origin = [0;0]
end;;

module ListPoint : POINT

For all intents and purposes, ListPoint is indistinguishable from PairPoint. For example:

---

5We could also implement getX and getY using pattern matching, as in let getX [x;_] = x, but this would generate warnings about non-exhaustive pattern matching.
The OCAML type system prevents abstraction violations on abstract types by assuming that two distinct abstract types are not equal. For example:

```
# let p2 = ListPoint.make 1 2;;
val p2 : ListPoint.point = <abstr>
# ListPoint.getX p2;;
- : int = 1
# ListPoint.getY p2;;
- : int = 2
# ListPoint.origin;;
- : ListPoint.point = <abstr>
# (ListPoint.getX ListPoint.origin, ListPoint.getY ListPoint.origin);;
- : int * int = (0, 0)
```

It is even possible to represent a point as a function:

```
# module PredPoint : POINT = struct
  type point = bool -> int
  let make x y = fun b -> if b then x else y
  let getX p = p true
  let getY p = p false
  let origin b = 0
end;;
module PairPoint : POINT
```

Functional points behave indistinguishably from other points:

```
# let p3 = PredPoint.make 1 2;;
val p3 : PredPoint.point = <abstr>
# PredPoint.getX p3;;
- : int = 1
# PredPoint.getY p3;;
- : int = 2
# PredPoint.origin;;
- : PredPoint.point = <abstr>
# (PredPoint.getX PredPoint.origin, PredPoint.getY PredPoint.origin);;
- : int * int = (0, 0)
```

### 4.2 Example: Environments

A more compelling example of an abstract data type is the mergeable environment datatype in Fig. 3. An environment is an abstraction that associates names with values. As we shall see later, interpreters, type checkers, and compilers all use environments for tracking the names used in a program.
The figure shows two implementations of the \textit{MENV} signature: \texttt{ListMenv}, which represents environments as lists of name/value pairs, and \texttt{FunMenv}, which represents environments as functions that map names to values. Below is a transcript of some interactions involving \texttt{ListMenv}:

\begin{verbatim}
# open ListMenv;;
# let e0 = empty;; (* The empty env *)
val e0 : 'a ListMenv.menv = <abstr>
# let e1 = bind "a" 1 e0;; (* The env a |-> 1 *)
val e1 : int ListMenv.menv = <abstr>
# let e2 = bind "b" 2 e1;; (* The env a |-> 1, b |-> 2 *)
val e2 : int ListMenv.menv = <abstr>
# let e3 = bind "a" 3 e2;; (* The env a |-> 3, b |-> 2 *)
val e3 : int ListMenv.menv = <abstr>
# let e4 = make ["b";"c";"d"] [4;5;6];; (* The env b |-> 4, c |-> 5, d |-> 6 *)
val e4 : int ListMenv.menv = <abstr>
# let e5 = merge e3 e4;; (* The env a |-> 3, b |-> 2, c |-> 5, d |-> 6 *)
val e5 : int ListMenv.menv = <abstr>
# let e6 = merge e4 e3;; (* The env a |-> 3, b |-> 4, c |-> 5, d |-> 6 *)
val e6 : int ListMenv.menv = <abstr>
# let envs = [e0;e1;e2;e3;e4;e5;e6]
val envs : int ListMenv.menv list =
  [<abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>]
# ListUtils.map (fun e -> lookup "a" e) envs;;
- : int option list = [None; Some 1; Some 1; Some 3; None; Some 3; Some 3]
# ListUtils.map (fun e -> lookup "b" e) envs;;
- : int option list = [None; None; Some 2; Some 2; Some 4; Some 2; Some 4]
# ListUtils.map (fun e -> lookup "c" e) envs;;
- : int option list = [None; None; None; None; Some 5; Some 5; Some 5]
# ListUtils.map (fun e -> lookup "d" e) envs;;
- : int option list = [None; None; None; None; Some 6; Some 6; Some 6]
# ListUtils.map (fun e -> lookup "e" e) envs;;
- : int option list = [None; None; None; None; None; None; None]
\end{verbatim}

An identical transcript would be obtained if every occurrence of \texttt{ListMenv} were replaced by \texttt{FunMenv}.

An interesting feature of the \texttt{FunMenv} module is that it uses functions as the data structure for implementing environments. In the First-Class Functions handout (#17), we saw that functions could be used to implement pairs (Church pairs) and natural numbers (Church numerals). The \texttt{FunMenv} module underscores that implementing data structures as functions is more than just an intellectual curiosity; it is a practical implementation technique! \texttt{FunMenv} is one of the many examples of functional implementation of data structures that we shall encounter in this course.

### 4.3 Example: Sets

A classic example of an ADT is a set. From the client’s perspective, a set is an abstract collection of values that contains each value at most once and which supports operations like membership testing, insertion, deletion, and the union, intersection, and difference of sets. An implementer can use any concrete data representation and algorithms to implement the set as long as the set operations work as expected. For example, the implementation may involve collections of elements potentially containing duplicate entries as long as the set functions make it appear as though the set contains exactly one occurrence of each element.

In \texttt{Ocaml} an ADT’s contract is represented as a signature and an ADT’s implementation is a module satisfying that signature. For example, Fig. 4 shows the signature for a set ADT. Each type declaration in the signature is accompanied by an English description of the meaning of the declared operation or value. By not giving a concrete definition of the \texttt{set} type, the declaration
module type MENV = sig
  type 'a menv
  val empty: 'a menv
  val bind : string -> 'a -> 'a menv -> 'a menv
  val make : string list -> 'a list -> 'a menv
  val lookup : string -> 'a menv -> 'a option
  val merge : 'a menv -> 'a menv -> 'a menv
end

module ListMenv : MENV = struct
  type 'a menv = (string * 'a) list
  let empty = []
  let bind name valu env = (name,valu) :: env
  let make names values = ListUtils.foldr2 bind empty names values
  let lookup name env =
    match ListUtils.some (fun (s,_) -> s = name) env with
    | None -> None
    | Some (_,valu) -> Some valu
  let merge env1 env2 = env1 @ env2
end

module FunMenv : MENV = struct
  type 'a menv = (string -> 'a option)
  let empty = fun s -> None
  let bind name valu env =
    fun s -> if s = name then Some valu else env s
  let make names values = ListUtils.foldr2 bind empty names values
  let lookup name env = env name
  let merge env1 env2 =
    fun s -> (match env1 s with
              | None -> env2 s
              | Some v -> Some v)
end

Figure 3: A signature and two implementations for mergeable environments.
type 'a set guarantees that the ADT is truly abstract. A client can only use the operations in
the signature to create and manipulate sets. The type system prevents any attempt by the client to
manipulate a set directly using knowledge of the underlying concrete representation. For instance,
if sets are represented as lists, then any attempt by the client to perform list operations directly on
a set will fail.

module type SET = sig

  type 'a set
  val empty : 'a set (* the empty set *)
  val singleton : 'a -> 'a set (* a set with one element *)
  val insert : 'a -> 'a set -> 'a set (* insert elt into given set *)
  val delete : 'a -> 'a set -> 'a set (* delete elt from given set *)
  val isEmpty: 'a set -> bool (* is the given set empty? *)
  val size : 'a set -> int (* number of distinct elements in given set *)
  val member : 'a -> 'a set -> bool (* is elt a member of given set? *)
  val union: 'a set -> 'a set -> 'a set (* union of two sets *)
  val intersection: 'a set -> 'a set -> 'a set (* intersection of two sets *)
  val difference: 'a set -> 'a set -> 'a set (* difference of two sets *)
  val fromList : 'a list -> 'a set (* create a set from a list *)
  val toList : 'a set -> 'a list (* list all set elts, sorted low to high *)
  val toString : ('a -> string) -> 'a set -> string (* string representation of the set *)

end

Figure 4: A signature for a set abstract data type (ADT).

The signature gives great latitude for an implementer to choose a representation for the ADT. In
the case of sets, a simple representation for a set is a list of elements without duplicates sorted from
low to high. For the ordering criteria, we use the built-in ordering that OCAML provides for any
type. A handy collection of functions for manipulating such lists is provided in the ListSetUtils
module (Fig. 5), whose signature is:

module type LIST_SET_UTILS = sig

  val member: 'a -> 'a list -> bool
  val insert: 'a -> 'a list -> 'a list
  val delete: 'a -> 'a list -> 'a list
  val union: 'a list -> 'a list -> 'a list
  val intersection: 'a list -> 'a list -> 'a list
  val difference: 'a list -> 'a list -> 'a list

end

A set implementation using these functions is the SortedListSet module presented in Fig. 6.
Of particular interest is the fromList function, which uses insert to insert all elements of the
given list into the resulting set. This preserves the invariant that the set must be a sorted list
without duplicates. It would be incorrect to defined fromList as

let fromList xs = xs

because the list xs might contain elements out of order or contain duplicate elements.

Of course, the SortedListSet module is only one possible implementation of the set ADT.
There are many other possible implementations, including variants of binary search trees (BSTs)
— binary trees of elements in which all elements in the left subtree of each node are strictly less
than the element value of that node, and all elements in the right subtree of each node are strictly
greater than the element value of that node. We will see how to implement binary trees in the next
module ListSetUtils : LIST_SET_UTILS = struct

  let rec member x ys =  
    match ys with  
      [] -> false  
    | y::ys' -> (x = y) || ((x > y) && (member x ys'))

  (* Insert an element into a sorted list *)
  let rec insert x ys =  
    match ys with  
      [] -> [x]  
    | y::ys' -> if x < y then x::ys  
                        else if x = y then ys  
                        else y::(insert x ys')

  (* Delete an element from a sorted list *)
  let rec delete x ys =  
    match ys with  
      [] -> []  
    | y::ys' -> if x = y then ys'  
                        else if x < y then ys  
                        else y::(delete x ys')

  (* Merge two sorted lists, removing duplicates *)
  let rec union xs ys =  
    match (xs, ys) with  
      ([], _) -> ys  
    | (_, []) -> xs  
    | (x::xs',y::ys') -> if x = y then x::(union xs' ys')  
                               else if x < y then x::(union xs' ys)  
                               else y::(union xs ys')

  (* Intersection of two sorted lists *)
  let rec intersection xs ys =  
    match (xs, ys) with  
      ([], _) -> []  
    | (_, []) -> []  
    | (x::xs',y::ys') -> if x = y then x::(intersection xs' ys')  
                                else if x < y then intersection xs' ys  
                                else intersection xs ys'

  (* Difference of two sorted lists *)
  let rec difference xs ys =  
    match (xs, ys) with  
      ([], _) -> []  
    | (_, []) -> xs  
    | (x::xs',y::ys') -> if x = y then difference xs' ys  
                               else if x < y then x::(difference xs' ys)  
                               else difference xs ys'

end

Figure 5: Utilities used to process sorted lists.
module SortedListSet : SET = struct

module LSU = ListSetUtils (* Abbreviation for list set utilities *)

type 'a set = 'a list

let empty = []

let singleton x = [x]

let insert x s = LSU.insert x s

let delete x s = LSU.delete x s

let isEmpty s = (s = [])

let size s = List.length s

let member x s = LSU.member x s

let union s1 s2 = LSU.union s1 s2

let intersection s1 s2 = LSU.intersection s1 s2

let difference s1 s2 = LSU.difference s1 s2

let toList s = s

let fromList xs = List.fold_right insert xs empty

let toString eltToString s = StringUtils.listToString eltToString s

end

Figure 6: An implementation of the set ADT using sorted lists.

lecture (Handout #25).

The OCAML type system is sophisticated enough to allow several implementations of the same ADT to be used in the same program. But it must be a type error for the operations of one implementation to be used on a value created by another implementation. For instance, suppose we have a BSTSet module implementing the SET signature in addition to the SortedListSet module, and we make the following two sets:

# let sls = SortedListSet.fromList [1;2;3];;
val sls : int SortedListSet.set = <abstr>

# let bst = BSTSet.fromList [2;3;4];;
val bst : int BSTSet.set = <abstr>

Then it should be a type error to use a SortedListSet operation on bst or to use a BSTSet operation on sls. And indeed it is:
OCAML is able to determine this by keeping track of which module the sets come from. In this case, \texttt{sls} has type \texttt{int SortedListSet.set}, while \texttt{bst} has type \texttt{int BSTSet.set}, and these types are considered distinct by the type system.

### 5 Functors

There are many situations where we would like to abstract over the particular structure that is used to implement a given signature. For example, we might want to implement some point functions (adding points, subtracting points, etc.) in terms of the \texttt{POINT} signature studied earlier. Because these operations can be written in terms of the abstract point operations, we want to be able to specify them in a way that is independent of the concrete representation of any particular implementation of the \texttt{POINT} signature.

Since structures are second-class entities in OCAML, we cannot use functions to abstract over them. However, OCAML supplies us with a function-like entity called a \texttt{functor} that is able to abstract over structures. In order to provide type safety and static linking guarantees, OCAML makes functors more restrictive than functions — they can only be declared and used in limited ways. Nevertheless, functors are still a powerful way to abstract over the details of particular structures.

Here is a simple example of a functor involving points:}

```ocaml
module PointOps = functor (P: POINT) -> struct
  let neg p = P.make (-P.getX p) (-P.getY p))
  let add p1 p2 = P.make ((P.getX p1) + (P.getX p2))
                  ((P.getY p1) + (P.getY p2))
  let sub p1 p2 = add p1 (neg p2)
  let toPair p = (P.getX p, P.getY p)
end
```

\texttt{PointOps} is a functor that takes as its single argument any structure \texttt{P} satisfying the \texttt{POINT} signature. As its result, it returns a structure that declares four point functions: a \texttt{neg} function that negates both coordinates of a point; an \texttt{add} function that performs componentwise addition of points; a \texttt{sub} function that performs componentwise subtraction of points; and a \texttt{toPair} function that converts a point to a pair.

The type of the above functor is:
module PointOps :
  functor (P : POINT) ->
  sig
    val neg : P.point -> P.point
    val add : P.point -> P.point -> P.point
    val sub : P.point -> P.point -> P.point
    val toPair : P.point -> int * int
  end

This type says that if the functor is applied to any structure \( P \) satisfying the \( \text{POINT} \) signature, the result is a structure containing the four functions \( \text{neg}, \text{add}, \text{sub}, \) and \( \text{toPair} \). The notation \( P\text{.point} \) indicates the point type that comes from argument, \( P \), given to the functor. A type that depends on the argument type to a functor is is known as a dependent type.

The \( \text{PointOps} \) functor can be applied to any structure satisfying the \( \text{POINT} \) signature. Here are some examples:

```ocaml
# module PairPointOps = PointOps(PairPoint);;
module PairPointOps :
  sig
    val neg : PairPoint.point -> PairPoint.point
    val add : PairPoint.point -> PairPoint.point -> PairPoint.point
    val sub : PairPoint.point -> PairPoint.point -> PairPoint.point
    val toPair : PairPoint.point -> int * int
  end

# PairPointOps.toPair (PairPointOps.sub (PairPointOps.make 8 1) (PairPointOps.make 3 4));;
- : int * int = (5, -3)
```

```ocaml
# module PredPointOps = PointOps(PredPoint);;
module PredPointOps :
  sig
    val neg : PredPoint.point -> PredPoint.point
    val add : PredPoint.point -> PredPoint.point -> PredPoint.point
    val sub : PredPoint.point -> PredPoint.point -> PredPoint.point
    val toPair : PredPoint.point -> int * int
  end

# PredPointOps.toPair (PredPointOps.sub (PredPointOps.make 8 1) (PredPointOps.make 3 4));;
- : int * int = (5, -3)
```

As another example of functors, suppose that we want to be able to write testing code for a set implementation that gives us confidence that the implementation is implemented correctly. Because we only care about the abstract behavior of sets in our testing code, we would like to be able to use the same testing code with any set implementation, regardless of its concrete representation.

We can achieve this goal using the set-testing functor \( \text{SimpleSetTest} \) shown in Fig. 7. This functor takes as its single argument any structure \( \text{Set} \) satisfying the \( \text{SET} \) signature. As its result, it returns a structure with the single declaration for a testing function named \( \text{test} \). This \( \text{test} \) function uses operations in the \( \text{Set} \) structure to manipulate sets of the type \( \text{int} \ \text{Set.set} \). It returns a triple of (1) a set containing the elements 1,2,4,5,6; (2) a string representation of this set; and (3) a list of integer lists showing the results of various set operations.

We can load \( \text{SimpleSetTest} \) into the top-level interpreter as follows:

```ocaml
# #use "../sets/SimpleSetTest.ml";;
module SimpleSetTest :
  functor (Set : SET) ->
  sig val test : unit -> int Set.set * string * int list list end
```

Note that \( \text{int} \ \text{Set.set} \) is a dependent type.
module SimpleSetTest = 
functor (Set: SET) -> struct
  let test () =
    let s1 = Set.fromList [5;2;6;1;4]
    and s2 = Set.fromList [2;8;6;3]
    in ( s1,
        Set.toString string_of_int s1,
        [Set.toList s1;
         Set.toList s2;
         Set.toList (Set.insert 3 s1);
         Set.toList (Set.delete 5 s1);
         Set.toList (Set.union s1 s2);
         Set.toList (Set.intersection s1 s2);
         Set.toList (Set.difference s1 s2)]
    )
  end
end

Figure 7: A simple set-testing functor.

We can now give SimpleSetTest a spin on different set structures:

# module SLST = SimpleSetTest(SortedListSet);;
module SLST : sig
  val test : unit -> int SortedListSet.set * string * int list list
end

# SLST.test();;
- : int SortedListSet.set * int list list * string list =
  ([<abstr>,
   "[1,2,4,5,6]",
   [[1; 2; 4; 5; 6]; [2; 3; 6; 8]; [1; 2; 3; 4; 5; 6]; [1; 2; 4; 6];
   [1; 2; 3; 4; 5; 6; 8]; [2; 6]; [1; 4; 5]],
  )

# module BSTST = SimpleSetTest(BSTSet);;
module BSTST : sig
  val test : unit -> int BSTSet.set * string * int list list
end

# BSTST.test();;
- : int BSTSet.set * int list list * string list =
  ([<abstr>,
    "((* 1 (* 2 *)) 4 ((* 5 *) 6 *))",
    [[1; 2; 4; 5; 6]; [2; 3; 6; 8]; [1; 2; 3; 4; 5; 6]; [1; 2; 4; 6];
    [1; 2; 3; 4; 5; 6; 8]; [2; 6]; [1; 4; 5]],
  )

By using the printed representation <abstr>, OCAML hides the implementation details of the given set structure. However, the toString function exposes the details of which structure is used in this example. This is not a failure of the OCAML module system; it just reflects that this operation is defined in an ambiguous way that allows it to return different results for different implementations.