12
Abstract data types and modular programming

The algebraic data types introduced in the last chapter are an expressive tool for defining sophisticated data structures. But with great power comes great responsibility.

As an example, consider one of the most fundamental of all data structures, the QUEUE. A queue is a collection of elements that admits of operations like creating an empty queue, adding elements one by one (called ENQUEUEING), and removing them one-by-one (called DEQUEUING), where crucially the first element enqueued is the first to be dequeued. The common terminology for this regimen is FIRST-IN-FIRST-OUT or FIFO.

We can provide a concrete implementation of the queue data type using the list data type, along with functions for enqueueing and dequeueing. An empty queue will be implemented as the empty list, with non-empty queues storing elements in order of their enqueueing, so newly enqueued elements are added at the end of the list.

```
# (* empty_queue -- An empty queue *)
# let empty_queue = [] ;;
val empty_queue : 'a list = []

# (* enqueue elt q -- Returns a queue resulting from
#    enqueuing a new elt onto q. *)
# let enqueue (elt : 'a) (q : 'a list) : 'a list =
#   q @ [elt] ;;
val enqueue : 'a -> 'a list -> 'a list = <fun>

# (* dequeue q -- Returns a pair of the next element
#    in q and the queue resulting from dequeueing
#    that element. *)
# let dequeue (q : 'a list) : 'a * 'a list =
#   match q with
#   | [] -> raise (Invalid_argument
#                    "dequeue: empty queue")
#   | hd :: tl -> hd, tl ;;
val dequeue : 'a list -> 'a * 'a list = <fun>
```

We can use these functions to enqueue and then dequeue a series of integers. Notice how the first element enqueued (the `1`) is the first element dequeued.

```
# let q = empty_queue
#         |> enqueue 1       (* enqueue 1, 2, and 4 *)
#         |> enqueue 2
#         |> enqueue 4 ;;
val q : int list = [1; 2; 4]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 1
val q : int list = [2; 4]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : int list = [4]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 4
val q : int list = []
```

Data structures built in this way can be used as intended, as they were above. (You'll note the FIFO behavior.) But if used in unexpected ways, things can go wrong quickly. Here, for instance, we enqueue some integers, then *reverse the queue* before dequeuing the elements in a *last*-in-first-out (LIFO) order. That's not supposed to happen.

```
# let q = empty_queue
#         |> enqueue 1       (* enqueue 1, 2, and 4 *)
#         |> enqueue 2
#         |> enqueue 4
#         |> List.rev ;;     (* yikes! *)
val q : int list = [4; 2; 1]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 4
val q : int list = [2; 1]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : int list = [1]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 1
val q : int list = []
```

Of course, reversing the elements is *not* an operation that ought to be possible on a queue. Queues, like other data structures, are defined by what operations can be performed on them, namely, enqueue and dequeue. These operations obey an INVARIANT, that the order in which elements appear when dequeued is the same as the order in which they were enqueued. Performing inappropriate operations on data structures is the path to violating such invariants, leading to software errors. Our implementation of queues as lists allows all sorts of inappropriate operations, like reversal of the enqueued elements, or taking the n-th element, or mapping over the elements, or any

other operation appropriate for lists but not queues. What we need is the ability to enforce restraint on the operations applicable to a data structure so as to preserve the invariants.

An analogy: The lights and heating in hotel rooms are intended to be on when the room is occupied, but they should be lowered when the room is empty. We can think of this as an invariant: If the room is unoccupied, the lights and heating are off. One approach to increasing compliance with this invariant is through documentation, placing a sign at the door "Please turn off the lights when you leave." But many hotels now use a key card switch, a receptacle near the door in which you insert the key card for the hotel room when you enter, in order to enable the lights and heating. (See Figure 12.1.) Since you have to bring your key card with you when you leave the room, thereby disabling the lights and heating, there is literally no way to violate the invariant. The state of California estimates that widespread use of hotel key card switches saves tens of millions of dollars per year (California Utilities Statewide Codes and Standards Team, 2011, page 6). Preventing violation of an invariant beats documenting it.

(a) (b)

Figure 12.1: Two approaches to preserving the invariant that the lights are off when the room is vacant: (a) an exhortation documenting the invariant; (b) a key card switch that disables the lights when the key is removed.

We've seen this idea of avoiding illegal states before in the edict of prevention. But in the queue example, type checking doesn't stop us from representing a bad state, and simple alternative representations for queues that prevent inappropriate operations don't come to mind. We need a way to implement new data types and operations such that the values of those types can only be used with the intended operations. We can't make the bad queues *unrepresentable*, but perhaps we can make them *inexpressible*, which should be sufficient for gaining the benefit of the edict of prevention.

The key idea is to provide an ABSTRACT DATA TYPE (ADT), a data type definition that provides not only a concrete IMPLEMENTATION of the data type values and operations on them, but also enforces that *only* those operations can be applied, making it impossible to express the application of other operations. This influential idea, the basis for modular programming, was pioneered by Barbara Liskov (Figure 12.2) in her CLU programming language.

Figure 12.2: The idea of abstract data types – grouping some functionality over types and hiding the implementation of that functionality behind a strict interface – is due to computer scientist Barbara Liskov, and is first seen in her influential CLU programming language from 1974. Her work on data abstraction and object-oriented programming led to her being awarded the 2008 Turing Award, computer science's highest honor.

The allowed operations are specified in a SIGNATURE; no other aspects of the implementation of the data type can be seen other than those specified by the signature. Users of the abstract data type can avail themselves of the functionality specified in the signature, while remaining oblivious of the particularities of the implementation. The signature specifies an *interface* to using the data structure, which serves as an ABSTRACTION BARRIER; only the aspects of the implementation specified in the signature may be made use of.

The idea of hiding aspects of the implementation from those who

shouldn't need access to those aspects is fundamental enough for an edict of its own, the edict of compartmentalization:

Edict of compartmentalization:
Limit information to those with a need to know.

In the case of the queue abstract data type, all that users of the implementation have a need to know is the types for the operations involving queues, viz., the creation of queues and the enqueueing and dequeueing of elements; that's all the signature should specify. The implementation may be in terms of lists (or any of a wide variety of other methods) but the users of the abstract data type should not be able to avail themselves of the further aspects of the implementation. By preventing them from using aspects of the implementation, the invariants implicit in the signature can be maintained. A further advantage of hiding the details of the implementation of a data structure behind the abstraction barrier (in addition to making illegal operations inexpressible) is that it becomes possible to modify the implementation without affecting its use. This aspect of abstract data types is tremendously powerful.

We've seen other applications of the edict of compartmentalization before, for instance, in the use of helper functions local to (and therefore only accessible to) a function being defined. The alternative, defining the helper function globally could lead to unintended use of and reliance on that function, which had been intended only for its more focused purpose.

12.1 Modules

In OCaml, abstract data types are implemented using MODULES. Modules provide a way of packaging together several components – types and values involving those types, including functions manipulating values of those types – subject to constraints of a signature. A module is specified by placing the definitions of its components between the keywords `struct` and `end`:

```
struct
  ⟨definition_1⟩
  ⟨definition_2⟩
  ⟨definition_3⟩
  ...
end
```

Each ⟨*definition*⟩ is a definition of a type or value (including functions, and even exceptions).

Just as values can be named using the `let` construct, modules can be named using the `module` construct:

`module` ⟨*modulename*⟩ `=`
⟨*moduledefinition*⟩

12.2 A queue module

As a first example of the use of modules to provide for abstract data types, we return to the queue data type that we started with, which provides a type for, say, integer queues, `int_queue`, together with functions `enqueue : int -> int_queue -> int_queue` and `dequeue : int_queue -> int * int_queue`. (Even better would be to generalize queues as polymorphically allowing for elements of any base type. We'll do so in Section 12.4.)

A module `IntQueue`[1] implementing the queue abstract data type is

```
# (* IntQueue -- An implementation of integer queues as
#    int lists, where the elements are kept with older
#    elements closer to the head of the list. *)
# module IntQueue =
#   struct
#     type int_queue = int list
#     let empty_queue : int_queue = []
#     let enqueue (elt : int) (q : int_queue)
#               : int_queue =
#       q @ [elt]
#     let dequeue (q : int_queue) : int * int_queue =
#       match q with
#       | [] -> raise (Invalid_argument
#                        "dequeue: empty queue")
#       | hd :: tl -> hd, tl
#   end ;;
module IntQueue :
  sig
    type int_queue = int list
    val empty_queue : int_queue
    val enqueue : int -> int_queue -> int_queue
    val dequeue : int_queue -> int * int_queue
  end
```

[1] Module names are required to begin with an uppercase letter. You've seen examples before in the `Stdlib` and `List` module names.

Exercise 101

Define a different implementation of integer queues as `int lists` where the elements are kept with older elements farther from the head of the list. What are the advantages and disadvantages of this implementation?

Components of a module are referenced using the already familiar notation of prefixing the module name and a dot before the component. We've seen this already in examples like `List.nth` or `Str.split`. Similarly, users of the `IntQueue` module can refer to `IntQueue.empty_queue` or `IntQueue.enqueue`. Let's use this module to perform various queue operations:

```
# let q = IntQueue.empty_queue
#         |> IntQueue.enqueue 1 (* enqueue 1, 2, and 4 *)
#         |> IntQueue.enqueue 2
#         |> IntQueue.enqueue 4 ;;
val q : IntQueue.int_queue = [1; 2; 4]
```

All of this module prefixing gets cumbersome quickly. We can instead just "open" the module to gain access to all of its components.[2]

```
# open IntQueue ;;
# let q = empty_queue
#         |> enqueue 1       (* enqueue 1, 2, and 4 *)
#         |> enqueue 2
#         |> enqueue 4 ;;
val q : IntQueue.int_queue = [1; 2; 4]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 1
val q : IntQueue.int_queue = [2; 4]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : IntQueue.int_queue = [4]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 4
val q : IntQueue.int_queue = []
```

Unfortunately, nothing restricts us from using arbitrary aspects of the module's implementation, for instance, reversing the elements of the queue.

```
# let q = empty_queue
#         |> enqueue 1       (* enqueue 1, 2, and 4 *)
#         |> enqueue 2
#         |> enqueue 4
#         |> List.rev   (* this shouldn't be allowed *) ;;
val q : int list = [4; 2; 1]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 4
val q : IntQueue.int_queue = [2; 1]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : IntQueue.int_queue = [1]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 1
val q : IntQueue.int_queue = []
```

What we need is a signature that restricts the use of the components of a module, just as a type restricts use of a value. This signature/module pairing carefully separates what the caller of code sees – the module signature, which provides the abstract type structure of the components, that is, how they are *used* – from what the implementer or developer sees – the module implementation, including the concrete types and values for the components, that is, how they are *implemented*.

[2] A useful technique to simplify access to a module without opening it (and thereby shadowing any existing names) is to provide a short alternative name for the module.

```
# module IQ = IntQueue ;;
module IQ = IntQueue
# let q = IQ.empty_queue
#         |> IQ.enqueue 1
#         |> IQ.enqueue 2
#         |> IQ.enqueue 4 ;;
val q : IQ.int_queue = [1; 2; 4]
```

Also of great utility is to open a module just within a particular local scope. OCaml provides for this with its LOCAL OPEN construct:

```
# let q =
#   let open IntQueue in
#   empty_queue
#   |> enqueue 1
#   |> enqueue 2
#   |> enqueue 4 ;;
val q : IntQueue.int_queue = [1; 2; 4]
```

The notation for specifying signatures is similar to that for modules, except for the use of `sig` instead of `struct`; and naming signatures is like naming modules with the addition of the evocative `type` keyword.

```
module type ⟨moduletype⟩ =
  sig
    ⟨definition1⟩
    ⟨definition2⟩
    ⟨definition3⟩
    ...
  end
```

We can define a signature `INT_QUEUE`[3] for an integer queue module:

[3] Signature names must also begin with an uppercase letter. We follow the stylistic convention of using all uppercase for signature names.

```
# module type INT_QUEUE =
#   sig
#     type int_queue
#     val empty_queue : int_queue
#     val enqueue : int -> int_queue -> int_queue
#     val dequeue : int_queue -> int * int_queue
#   end ;;
module type INT_QUEUE =
  sig
    type int_queue
    val empty_queue : int_queue
    val enqueue : int -> int_queue -> int_queue
    val dequeue : int_queue -> int * int_queue
  end
```

The signature provides a full listing of all the aspects of a module that are visible to users of the module. In particular, the module provides a type called `int_queue`, but since the concrete implementation of that type is not provided in the signature, it is unavailable to users of modules satisfying the signature. The signature states that the module must provide a value `empty_queue` but what the concrete implementation of that value is is again hidden. And so on.

Notice that where the module implementation defines named values using the `let` construct, the signature uses the `val` construct, which provides a name and a type, but no definition of what is named.

Extending the analogy between signatures and types further, we can specify that a module satisfies and is constrained by a signature with a notation almost identical to that constraining a value to a certain type.

```
module ⟨modulename⟩ : ⟨signature⟩ =
  ⟨moduledefinition⟩
```

We could define `IntQueue` as satisfying the `INT_QUEUE` signature by adding this kind of "typing" as in the highlighted addition below:

```
# (* IntQueue -- An implementation of integer queues as
#    int lists, where the elements are kept with older
```

```
#    elements closer to the head of the list. *)
# module IntQueue : INT_QUEUE =
#   struct
#     type int_queue = int list
#     let empty_queue : int_queue = []
#     let enqueue (elt : int) (q : int_queue)
#                 : int_queue =
#       q @ [elt]
#     let dequeue (q : int_queue) : int * int_queue =
#       match q with
#       | [] -> raise (Invalid_argument
#                        "dequeue: empty queue")
#       | hd :: tl -> hd, tl
#   end ;;
module IntQueue : INT_QUEUE
```

This module implements integer queues abstractly, allowing access only as specified by the `INT_QUEUE` signature. For instance, after building a queue, we no longer have access to its concrete implementation.

```
# open IntQueue ;;
# let q = empty_queue
#         |> enqueue 1       (* enqueue 1, 2, and 4 *)
#         |> enqueue 2
#         |> enqueue 4 ;;
val q : IntQueue.int_queue = <abstr>
```

The value of `q` is reported simply as `<abstr>` connoting an abstract value hidden behind the abstraction barrier. We can't "see inside". Similarly, application of an operation not sanctioned by the signature, like list reversal, now fails.

```
# List.rev q ;;
Line 1, characters 9-10:
1 | List.rev q ;;
             ^
Error: This expression has type IntQueue.int_queue
       but an expression was expected of type 'a list
```

OCaml reports a type error. The function `List.rev` requires an argument of type `'a list`, but it is being applied to a queue, of type `IntQueue.int_queue`. True, the type `IntQueue.int_queue` is implemented as an `'a list`, but that fact is hidden from users of the module by the signature, hidden behind the abstraction barrier.

12.3 Signatures hide extra components

What happens when a module defines more components than its signature provides for? As a trivial example, we will define an ORDERED TYPE as a type that has an associated comparison function that provides an ordering on elements of the type. The definition of such a

module provides for these two components: a type, call it `t`, and a function that takes two elements `x` and `y` of type `t` and returns an integer indicating the ordering of the two, `-1` if `x` is smaller, `+1` if `x` is larger, and `0` if the two are equal in the ordering.[4]

[4] We use this arcane approach for the `compare` function to mimic the `Stdlib.compare` library function. Frankly, a better approach would be to take the result of the comparison to be a value in an enumerated type defined as `type order = Less | Equal | Greater`.

This specification of what constitutes an ordered type can be captured in a signature `ORDERED_TYPE`:

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

A simple implementation of an ordered type is based on the string type. Notice that we explicitly specify the signature for the module:

```
# module StringOrderedType : ORDERED_TYPE =
#   struct
#     type t = string
#     let compare = Stdlib.compare
#   end ;;
module StringOrderedType : ORDERED_TYPE
```

We take advantage of the built in `compare` function in the `Stdlib` module,[5] which is a general purpose comparison function that uses the same return value convention of `-1`, `0`, `+1` for elements that are less than, equal, and greater than, respectively. A more interesting example is an ordered type for points (pairs of floats) where the ordering on points is based on which is closer to the origin. This time, however, we don't specify a signature for the module:

[5] Although the `Stdlib` prefix isn't needed – the components of the `Stdlib` module are always available – we add it here for clarity.

```
# module PointOrderedType =
#   struct
#     type t = float * float
#     let norm (x, y) =
#       x ** 2. +. y ** 2.
#     let compare p1 p2 =
#       Stdlib.compare (norm p1) (norm p2)
#   end ;;
module PointOrderedType :
  sig
    type t = float * float
    val norm : float * float -> float
    val compare : float * float -> float * float -> int
  end
```

We can make use of the module to see how this ordering works on some examples.

```
# let open PointOrderedType in
#   compare (1., 1.) (5., 0.),
```

```
#   compare (1., 1.) (-1., -1.),
#   compare (1., 1.) (0., 1.1) ;;
- : int * int * int = (-1, 0, 1)
```

Note that the `PointOrderedType` module contains three components: the type `t`, and functions `norm` and `compare`. It goes beyond the `ORDERED_TYPE` signature in providing an extra function,

```
# PointOrderedType.norm ;;
- : float * float -> float = <fun>
# PointOrderedType.norm (1., 1.) ;;
- : float = 2.
```

since we did not explicitly restrict it to that signature. If instead we restrict `PointOrderedType` to the `ORDERED_TYPE` signature, only the components in that signature are made available.

```
# module PointOrderedType : ORDERED_TYPE =
#   struct
#     type t = float * float
#     let norm (x, y) =
#       x ** 2. +. y ** 2.
#     let compare p1 p2 =
#       Stdlib.compare (norm p1) (norm p2)
#   end ;;
module PointOrderedType : ORDERED_TYPE
```

The `norm` function is no longer defined:

```
# PointOrderedType.norm ;;
Line 1, characters 0-21:
1 | PointOrderedType.norm ;;
    ^^^^^^^^^^^^^^^^^^^^^
Error: Unbound value PointOrderedType.norm
```

In general, *only the aspects of a module consistent with its signature are visible outside of its implementation* to users of the module. All other aspects are hidden behind the abstraction barrier. In particular, the `norm` function is not available, and the identity of the type `t` is hidden as well. We can tell, because we no longer can compare two points.

```
# PointOrderedType.compare (1., 1.) (5., 0.) ;;
Line 1, characters 25-33:
1 | PointOrderedType.compare (1., 1.) (5., 0.) ;;
                             ^^^^^^^^
Error: This expression has type 'a * 'b
       but an expression was expected of type PointOrderedType.t
```

The arguments we are providing are expected to be of type `t` but we are providing arguments of type `float * float`. Although the implementation equates these types, outside of the abstraction barrier their equality isn't known. (Yes, this is a problem. We'll address it using sharing constraints later in Section 12.5.2.)

A fundamental role of modules and their signatures is to establish these abstraction barriers so that information about how data types happen to be implemented can't leak out and be taken advantage of.

12.4 Modules with polymorphic components

Returning to the queue example, there's no reason to restrict queues to integer elements. We can make the components of the module polymorphic, using type variables as usual to capture the places where arbitrary types can appear. We start with a polymorphic queue signature:

```
# module type QUEUE = sig
#   type 'a queue
#   val empty_queue : 'a queue
#   val enqueue : 'a -> 'a queue -> 'a queue
#   val dequeue : 'a queue -> 'a * 'a queue
# end ;;
module type QUEUE =
  sig
    type 'a queue
    val empty_queue : 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val dequeue : 'a queue -> 'a * 'a queue
  end
```

and define a queue module satisfying the signature:

```
# (* Queue -- An implementation of polymorphic queues
#    as lists, where the elements are kept with older
#    elements closer to the head of the list. *)
# module Queue : QUEUE = struct
#   type 'a queue = 'a list
#   let empty_queue : 'a queue = []
#   let enqueue (elt : 'a) (q : 'a queue) : 'a queue =
#     q @ [elt]
#   let dequeue (q : 'a queue) : 'a * 'a queue =
#     match q with
#     | [] -> raise (Invalid_argument
#                      "dequeue: empty queue")
#     | hd :: tl -> hd, tl
# end ;;
module Queue : QUEUE
```

Now we can avail ourselves of queues of different types:

```
# open Queue ;;
# let intq = empty_queue
#            |> enqueue 1
#            |> enqueue 2 ;;
val intq : int Queue.queue = <abstr>
# let boolq = empty_queue
#             |> enqueue true
```

```
#             |> enqueue false ;;
val boolq : bool Queue.queue = <abstr>
# dequeue intq ;;
- : int * int Queue.queue = (1, <abstr>)
# dequeue boolq ;;
- : bool * bool Queue.queue = (true, <abstr>)
```

Exercise 102

In Section 11.3, we provided a data type for dictionaries that makes sure that the keys and values match up properly. We noted, however, that nothing prevents building a dictionary with multiple occurrences of the same key.

Define a dictionary module signature and implementation that implements dictionaries using the type from Section 11.3, and provides a function

```
add : ('key, 'value) dictionary -> 'key -> 'value ->
('key, 'value) dictionary
```

for adding a key and its value to a dictionary, and a function

```
lookup : ('key, 'value) dictionary -> 'key -> 'value
option
```

for looking keys up in the dictionary. The `add` function should raise an appropriate exception if the key being added already appears in the dictionary. The `lookup` function should return `None` if the key being looked up does not appear in the dictionary. The signature should hide the implementation of the type and the functions so that the only access to the dictionary is through these two functions.

Can you express a dictionary built using this module that has duplicate keys?

12.5 *Abstract data types and programming for change*

One of the primary advantages of using abstract data types (as opposed to concrete data structures) is that by hiding the data type implementations, the implementations can be changed without affecting users of the data types.

Recall the `query` type from Section 11.2.

```
# type query =
#   | Word of string
#   | And of query * query
#   | Or of query * query ;;
type query = Word of string | And of query * query | Or of query *
  query
```

In that section, a corpus of documents was structured as a list of pairs, each containing a name and a list of strings, the words in the document. Given that we'll be searching for particular words in documents, an alternative data structure useful for search is the REVERSE INDEX, a kind of dictionary with words as the keys and a set of document identifiers (the title strings, say) as the values.

If we implement this concretely, using a list of pairs for the dictionary and a string list for the set of document titles, we end up with the following type:

```
# type index = (string * (string list)) list ;;
type index = (string * string list) list
```

Using a reverse index, the code for evaluating a query is quite simple:

```
# let rec eval (q : query)
#              (idx : index)
#            : string list =
#   match q with
#   | Word word ->
#      let (_key, targets) =
#        List.find (fun (w, _lst) -> w = word) idx
#      in targets
#   | And (q1, q2) ->
#      intersection (eval q1 idx) (eval q2 idx)
#   | Or (q1, q2) ->
#      (eval q1 idx) @ (eval q2 idx) ;;
Line 10, characters 0-12:
10 | intersection (eval q1 idx) (eval q2 idx)
     ^^^^^^^^^^^^
Error: Unbound value intersection
```

Of course, we'll need code for the intersection of two lists. Here's an approach, in which the lists are kept sorted to facilitate finding duplicates:

```
# let rec intersection set1 set2 =
#   match set1, set2 with
#   | [], _
#   | _, [] -> []
#   | h1 :: t1, h2 :: t2 ->
#       if h1 = h2 then h1 :: intersection t1 t2
#       else if h1 < h2 then intersection t1 set2
#       else intersection set1 t2 ;;
val intersection : 'a list -> 'a list -> 'a list = <fun>
```

Now, we might get lucky and notice a problematic clash of assumptions in the `eval` function. The `intersection` function assumes the lists are sorted, but the final match in `eval` just appends two lists to form the union of their elements. Nothing guarantees that the result of the union is sorted. We can fix that up by using a sort function from the `List` module.

```
# let rec eval (q : query)
#              (idx : index)
#            : string list =
#   match q with
#   | Word word ->
#      let (_, targets) =
#        List.find (fun (w, _lst) -> w = word) idx
#      in targets
#   | And (q1, q2) ->
#      intersection (eval q1 idx) (eval q2 idx)
#   | Or (q1, q2) ->
```

```
#      List.sort_uniq compare
#        ((eval q1 idx) @ (eval q2 idx)) ;;
val eval : query -> index -> string list = <fun>
```

But maybe then we notice that in our application, this `List.find` lookup takes too much time. It has to look through the elements of the list sequentially to find the one for the word we're looking up. That takes time proportional to the number of words being indexed. (More on this kind of issue in Chapter 14.) Maybe you recall from an earlier course that hash tables allow lookup in constant time, and you think to use them. Luckily, the `Hashtbl` library module provides hash tables. To incorporate hash tables, we have to change the `index` type:

```
# type index = (string, string list) Hashtbl.t ;;
type index = (string, string list) Hashtbl.t
```

as well as the word query lookup:

```
# let rec eval (q : query)
#              (idx : index)
#            : string list =
#   match q with
#   | Word word -> Hashtbl.find idx word
#   | And (q1, q2) ->
#      intersection (eval q1 idx) (eval q2 idx)
#   | Or (q1, q2) ->
#      List.sort_uniq compare
#        ((eval q1 idx) @ (eval q2 idx)) ;;
val eval : query -> index -> string list = <fun>
```

There's a theme here. *Every change to the underlying data representation requires multiple changes to the code*, even though nothing has changed conceptually in the underlying use of the data. We're still searching in the data, taking unions and intersections.

Let's go back to the original specification of the reverse index: "a kind of dictionary with words as the keys and a set of document identifiers (the title strings, say) as the values." This specification talks about abstract data types like dictionaries and sets, but we've been trying to directly implement them in terms of lists and pairs and hash tables. By embracing the abstractions, we can hide all of the details from our indexing code.

Suppose we had modules for string sets and for indexes. The string set module, call it `StringSet`, would presumably provide set functions like `union` and `intersection`. The index module, call it `Index` would provide a lookup `function`. The `eval` function using these modules then becomes

```
let rec eval (q : query)
             (idx : Index.dict)
           : StringSet.set =
```

```
match q with
| Word word -> (match Index.lookup idx word with
                 | None -> StringSet.empty
                 | Some v -> v)
| And (q1, q2) -> StringSet.intersection (eval q1 idx)
                                          (eval q2 idx)
| Or (q1, q2) -> StringSet.union (eval q1 idx)
                                 (eval q2 idx) ;;
```

This is much nicer. It says what the code does *at the right level of abstraction*, in terms of high-level operations like dictionary lookup, or set intersection and union. It remains silent, as it should, about exactly how those operations are implemented.

Now we'll need module definitions for `Index` and `StringSet`. We start with `StringSet` first, and in particular, its module signature, since this specifies how the module can be used.

12.5.1 A string set module

A string set module needs to provide some operations for creating and manipulating the sets. The requirements can be specified in a module signature. Here's a first cut:

```
# module type STRING_SET =
#   sig
#     (* Type of string sets *)
#     type set
#     (* An empty set *)
#     val empty : set
#     (* Returns true if set is empty, false otherwise *)
#     val is_empty : set -> bool
#     (* Adds string to existing set (if not already a member) *)
#     val add : string -> set -> set
#     (* Union of two sets *)
#     val union : set -> set -> set
#     (* Intersection of two sets *)
#     val intersection : set -> set -> set
#     (* Returns true iff string is in set *)
#     val member: string -> set -> bool
#   end ;;
module type STRING_SET =
  sig
    type set
    val empty : set
    val is_empty : set -> bool
    val add : string -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : string -> set -> bool
  end
```

Any implementation of this signature must provide:

- a type, called `set`;
- an element of that type called `empty`;
- a function that maps elements of the type to bool, called `is_empty`;
- and so forth.

From the point of view of the users (callers) of this abstract data type, this is all they need to know: The name of the type and the functions that apply to values of that type.

To drive this point home, we'll make use of an implementation (`StringSet`) of this abstract data type before even looking at the implementing code.

```
# let s = StringSet.add "c"
#           (StringSet.add "b"
#              (StringSet.add "a" StringSet.empty)) ;;
val s : StringSet.set = <abstr>
```

Note that the string set we've called `s` is of the abstract type `StringSet.set` and the particulars of the value implementing the set are hidden from us as `<abstr>`.

The types, values, and functions provided in the signature are normal OCaml objects that interact with the rest of the language as usual. We can still avail ourselves of the rest of OCaml. For instance, we can clean up the definition of `s` using reverse application and a local open:

```
# let s =
#   let open StringSet in
#   empty
#   |> add "a"
#   |> add "b"
#   |> add "c" ;;
val s : StringSet.set = <abstr>
```

Other operations work as well.

```
# StringSet.member "a" s ;;
- : bool = true
# StringSet.member "d" s ;;
- : bool = false
```

Of course, the ADT must have an actual implementation for it to work. We've just been assuming one, but we can provide a possible implementation (the one we've been using as it turns out), obeying the specific signature we just defined.[6]

```
module StringSet : STRING_SET =
  (* Implementation of STRING_SET as list of strings.
     Assumes list may be unsorted but with no duplicates. *)
  struct
```

[6] You'll notice that we don't bother adding types to the definitions of the values in this module implementation. Since the signature already provided explicit types (satisfying the edict of intention), OCaml can verify that the implementation respects those types. Nonetheless, it can sometimes be useful to provide further typing information in a module implementation.

```
  type set = string list
  let empty = []
  let is_empty set = (set = [])
  let member = List.mem
  let add elt set =
    if List.mem elt set then set
    else elt :: set
  let union = List.fold_right add
  let rec intersection set1 set2 =
    match set1 with
    | [] -> []
    | hd :: tl -> let tlint = intersection tl set2 in
                  if member hd set2 then add hd tlint
                  else tlint
end ;;
```

In this implementation, sets are implemented as string lists. A comment documents the invariant in the implementation that the lists have no duplicates, though they might not be sorted. But there's no way for a user of this module to know any of that; the signature doesn't reveal anything about the implementation type. Even though the sets are implemented as string lists, if we try to do string-list-like operations, we'll be thwarted.

```
# s @ ["b"; "e"] ;;
Line 1, characters 0-1:
1 | s @ ["b"; "e"] ;;
    ^
Error: This expression has type StringSet.set
       but an expression was expected of type 'a list
```

And it's a good thing too, because if we could have added the `"b"` to the list, suddenly, the list doesn't obey the invariant required by the implementation that there be no duplicates. But because of the abstraction barrier, there's no way for a user of the module to break the invariant, so long as the implementation maintains it.

Because the sets are implemented as unsorted lists, when taking the union of two sets `set1` and `set2`, we must traverse the entirety of the `set2` list once for each element of `set1`. For small sets, this is not likely to be problematic, and worrying about this inefficiency may well be a premature effort at optimization.[7] But for a set implementation likely to be used widely and on very large sets, it may be useful to address the issue.

[7] In the introduction to Chapter 14 you'll learn that "premature optimization is the root of all evil."

A better alternative from an efficiency point of view is to implement sets as sorted lists. This requires a bit more work in adding elements to a set to place them in the right order, but saves effort for union and intersection. We redefine the `StringSet` module accordingly, still satisfying the same `STRING_SET` signature.

```
# (* Implementation of STRING_SET as list of strings.
```

```
#    Assumes list is *sorted* with no duplicates. *)
# module StringSet : STRING_SET =
#   struct
#     type set = string list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl -> if elt = hd then true
#                     else if elt < hd then false
#                     else member elt tl
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl -> if elt < hd then elt :: s
#                     else if elt = hd then s
#                     else hd :: add elt tl
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#           if h1 = h2 then h1 :: intersection t1 t2
#           else if h1 < h2 then intersection t1 set2
#           else intersection set1 t2
#   end ;;
module StringSet : STRING_SET
```

Now we can test the revised definition.

```
# let s =
#   let open StringSet in
#   empty
#   |> add "a"
#   |> add "b"
#   |> add "c" ;;
val s : StringSet.set = <abstr>
# StringSet.member "a" s ;;
- : bool = true
# StringSet.member "d" s ;;
- : bool = false
```

And here's the payoff. Even though we've completely changed the implementation of string sets, even using a data structure obeying a different invariant, the code for using string sets changes *not at all*.

12.5.2 A generic set signature

For document querying, we needed a string set module. For other purposes we may need sets of other element types. We could generate similar modules for, say, integer sets, with an appropriate signature:

```
# module type INT_SET =
#   sig
#     (* Type of integer sets *)
#     type set
#     (* The empty set *)
#     val empty : set
#     (* Returns true if set is empty; false otherwise *)
#     val is_empty : set -> bool
#     (* Adds integer to existing set (if not already a member) *)
#     val add : int -> set -> set
#     (* Union of two sets *)
#     val union : set -> set -> set
#     (* Intersection of two sets *)
#     val intersection : set -> set -> set
#     (* Returns true iff integer is in set *)
#     val member: int -> set -> bool
#   end ;;
module type INT_SET =
  sig
    type set
    val empty : set
    val is_empty : set -> bool
    val add : int -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : int -> set -> bool
  end
```

but we'd be violating the edict of irredundancy. Rather, we'd prefer a generic signature for set modules that provides a set type for any element type.

Here is such a signature. We've added a new type to the module, the type `element` for elements of the set, and we use it in the types of the various functions.

```
# module type SET =
#   sig
#     (* Type of sets *)
#     type set
#     (* and their elements *)
#     type element
#     (* The empty set *)
#     val empty : set
#     (* Returns true if set is empty; false otherwise *)
#     val is_empty : set -> bool
#     (* Adds element to existing set (if not already a member) *)
#     val add : element -> set -> set
#     (* Union of two sets *)
#     val union : set -> set -> set
#     (* Intersection of two sets *)
#     val intersection : set -> set -> set
#     (* Returns true iff element is in set *)
#     val member: element -> set -> bool
```

```
#   end ;;
module type SET =
  sig
    type set
    type element
    val empty : set
    val is_empty : set -> bool
    val add : element -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : element -> set -> bool
  end
```

A string set implementation satisfying this signature defines the `element` type as `string`:

```
# module StringSet : SET =
#   struct
#     type element = string
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl -> if elt = hd then true
#                     else if elt < hd then false
#                     else member elt tl
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl -> if elt < hd then elt :: s
#                     else if elt = hd then s
#                     else hd :: add elt tl
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#           if h1 = h2 then h1 :: intersection t1 t2
#           else if h1 < h2 then intersection t1 set2
#           else intersection set1 t2
#   end ;;
module StringSet : SET
```

We can use this `StringSet` to, for instance, generate an empty string set:

```
# StringSet.empty ;;
- : StringSet.set = <abstr>
```

We run into a major problem, though, in the simple act of checking if a string is a member of the set.

```
# StringSet.member "a" StringSet.empty ;;
Line 1, characters 17-20:
1 | StringSet.member "a" StringSet.empty ;;
                     ^^^
Error: This expression has type string but an expression was
  expected of type
         StringSet.element
```

What's the problem? It turns out that the abstraction barrier provided by the SET signature is doing exactly what it should. The implementation promises to deliver something that satisfies and reveals SET. And that's all. The SET signature reveals types `set` and `element`, not `string list` and `string`. Viewed from within the implementation, the types `element` and `string` are the same. But from outside the module implementation, only `element` is available, leading to the type mismatch with `string`.

This is a case in which the abstraction barrier is too strict. (We saw this before in Section 12.3.) We do want to allow the user of the module to have access to the implementation of the `element` type, if only so that module users can provide elements of that type. Rather than using the too abstract SET signature, we can define slightly less abstract signatures using SHARING CONSTRAINTS, which augment a signature with one or more type equalities across the abstraction barrier, identifying abstract types within the signature (`element`) with implementations of those types accessible outside the implementation (`string`).[8]

```
# module type STRING_SET = SET with type element = string ;;
module type STRING_SET =
  sig
    type set
    type element = string
    val empty : set
    val is_empty : set -> bool
    val add : element -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : element -> set -> bool
  end
```

Now we can declare the implementation as satisfying this relaxed signature.

```
# module StringSet : STRING_SET =
#   struct
#     type element = string
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
```

[8] Notice how in printing out the result of defining the new `STRING_SET` signature, OCaml specifies that the type of elements is `string`. Compare this with the version above without the sharing constraint.

This example requires only a single sharing constraint, but multiple constraints can be useful as well. They are combined with the `and` keyword, for example, the pair of sharing constraints `with type key = D.key and type value = D.value` used in the definition of the `MakeOrderedDict` module in Section 12.6.

```
#       | [] -> false
#       | hd :: tl -> if elt = hd then true
#                     else if elt < hd then false
#                     else member elt tl
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl -> if elt < hd then elt :: s
#                     else if elt = hd then s
#                     else hd :: add elt tl
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#           if h1 = h2 then h1 :: intersection t1 t2
#           else if h1 < h2 then intersection t1 set2
#           else intersection set1 t2
#   end ;;
module StringSet : STRING_SET
```

This implementation now allows us to perform operations involving particular strings.

```
# StringSet.empty ;;
- : StringSet.set = <abstr>
# StringSet.member "a" StringSet.empty ;;
- : bool = false
# let s =
#   let open StringSet in
#     empty
#     |> add "first"
#     |> add "second"
#     |> add "third" ;;
val s : StringSet.set = <abstr>
# StringSet.union s s ;;
- : StringSet.set = <abstr>
# StringSet.member "a" s ;;
- : bool = false
```

12.5.3 A generic set implementation

Sharing constraints solve the problem of duplicative signatures, because we can define different signatures by adding different sharing constraints to the generic SET signature:

```
# module type STRING_SET =
#   SET with type element = string ;;
# module type INT_SET =
#   SET with type element = int ;;
# module type INTBOOL_SET =
#   SET with type element = int * bool ;;
```

Unfortunately, they do nothing for the problem of duplicative implementations. To implement a module satisfying the `INT_SET` signature, we'd need to build the whole module from scratch, like this:

```
# module IntSet : INT_SET =
#   struct
#     type element = int
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl -> if elt = hd then true
#                     else if elt < hd then false
#                     else member elt tl
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl -> if elt < hd then elt :: s
#                     else if elt = hd then s
#                     else hd :: add elt tl
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#           if h1 = h2 then h1 :: intersection t1 t2
#           else if h1 < h2 then intersection t1 set2
#           else intersection set1 t2
#   end ;;
```

The redundancy is massive; the only differences from the `StringSet` implementation are those highlighted in red. To solve this violation of the edict of irredundancy requires more powerful tools.

What we need is a way of *generating* implementations that depend on some stuff. In the case at hand, the stuff is just the implementation of the `element` type, and perhaps some functionality involving that type. For instance, in the implementations of the `StringSet` and `IntSet` modules, we availed ourselves of comparing elements using the `<` operator. Any type we build a set from using this implementation approach needs some way of performing such comparisons, but the `<` operator may not always be appropriate for that purpose. More generally, the implementations may depend not only on a type but on some values of that type or functions over the type, or even multiple types.

If only we had a way of packaging up some types and related values and functions. But we do have such a way: the module system itself. In effect, what we need is something akin to a function that takes as ar-

gument a module defining the parameters of the implementation and returns the desired module. We call these "functions" from modules to modules FUNCTORS.

We can use the `StringSet` and `IntSet` implementations as the basis for a functor `MakeOrderedSet`, which takes a module as argument to provide the element type and returns a module satisfying the `SET` signature. As described above, the argument module should have a type (call it `t`) and a way of comparing elements of the type (call it `compare`). We'll have the `compare` function take two elements of type `t` and return an integer specifying whether the first integer is less than (`-1`), equal to (`0`), or greater than (`1`) the second integer.

You may recognize this signature. It's the `ORDERED_TYPE` signature from Section 12.3, repeated here for reference.

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

The argument to the functor should satisfy this signature.

A functor that takes a module with this signature and delivers a `SET` implementation can be constructed just by factoring out the type and the comparison from our previous implementations of `IntSet` and `StringSet`.

```
# module MakeOrderedSet (Elements : ORDERED_TYPE) : SET =
#   struct
#     type element = Elements.t
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl ->
#           (match Elements.compare elt hd with
#            | 0  (* equal *)   -> true
#            | -1 (* less *)    -> false
#            | _  (* greater *) -> member elt tl)
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl ->
#           (match Elements.compare elt hd with
#            | 0  (* equal *)   -> s
#            | -1 (* less *)    -> elt :: s
#            | _  (* greater *) -> hd :: add elt tl)
#     let union = List.fold_right add
```

```
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#           (match Elements.compare h1 h2 with
#            | 0  (* equal *)   -> h1 :: intersection t1 t2
#            | -1 (* less *)    -> intersection t1 set2
#            | _  (* greater *) -> intersection set1 t2)
#   end ;;
module MakeOrderedSet : functor (Elements : ORDERED_TYPE) -> SET
```

But this won't do. The returned module satisfies `SET`, but we've already seen how this is too strong a requirement. The solution is the same as before, use sharing constraints to allow access to the `element` type.

```
# module MakeOrderedSet (Elements : ORDERED_TYPE)
#          : (SET with type element = Elements.t) =
#   struct
#     type element = Elements.t
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl ->
#           (match Elements.compare elt hd with
#            | 0  (* equal *)   -> true
#            | -1 (* less *)    -> false
#            | _  (* greater *) -> member elt tl)
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl ->
#           (match Elements.compare elt hd with
#            | 0  (* equal *)   -> s
#            | -1 (* less *)    -> elt :: s
#            | _  (* greater *) -> hd :: add elt tl)
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#           (match Elements.compare h1 h2 with
#            | 0  (* equal *)   -> h1 :: intersection t1 t2
#            | -1 (* less *)    -> intersection t1 set2
#            | _  (* greater *) -> intersection set1 t2)
#   end ;;
module MakeOrderedSet :
  functor (Elements : ORDERED_TYPE) ->
    sig
```

```
    type set
    type element = Elements.t
    val empty : set
    val is_empty : set -> bool
    val add : element -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : element -> set -> bool
  end
```

Here we finally have a functor that can generate a set module for any type. Let's generate a few, starting with a string set module, which we can generate by applying the `MakeOrderedSet` functor to a module satisfying `ORDERED_TYPE` linking the `string` type to an appropriate ordering function (here, the default `Stdlib.compare` function).

```
# module StringSet = MakeOrderedSet
#                      (struct
#                         type t = string
#                         let compare = compare
#                       end) ;;
module StringSet :
  sig
    type set
    type element = string
    val empty : set
    val is_empty : set -> bool
    val add : element -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : element -> set -> bool
  end
```

It works as expected:

```
# let s =
#   let open StringSet in
#     empty
#     |> add "first"
#     |> add "second"
#     |> add "third" ;;
val s : StringSet.set = <abstr>
# StringSet.union s s ;;
- : StringSet.set = <abstr>
# StringSet.member "a" s ;;
- : bool = false
```

How about an integer set module? Again, a couple of lines of code suffice.

```
# module IntSet = MakeOrderedSet
#                   (struct
#                      type t = int
#                      let compare = compare
```

```
#                   end) ;;
module IntSet :
  sig
    type set
    type element = int
    val empty : set
    val is_empty : set -> bool
    val add : element -> set -> set
    val union : set -> set -> set
    val intersection : set -> set -> set
    val member : element -> set -> bool
  end
# let s =
#   let open IntSet in
#     empty
#     |> add 1
#     |> add 2
#     |> add 3 ;;
val s : IntSet.set = <abstr>
# IntSet.union s s ;;
- : IntSet.set = <abstr>
# IntSet.member 4 s ;;
- : bool = false
```

12.6 A dictionary module

The query evaluation application we've been working on (remember that?) required not only an implementation of a set ADT, but also a dictionary ADT. Dictionaries are data structures that associate keys to values, and allow for insertion and deletion of key-value associations, and looking up of the value associated with a given key (if one exists).

We now have all the tools to build that as well. An appropriate signature for a dictionary is

```
# module type DICT =
#   sig
#     type key
#     type value
#     type dict
#
#     (* empty -- An empty dictionary *)
#     val empty : dict
#     (* lookup dict key -- Returns as an option the value
#        associated with the provided key. If the key is
#        not in the dictionary, returns None. *)
#     val lookup : dict -> key -> value option
#     (* member dict key -- Returns true if and only if the
#        key is in the dictionary. *)
#     val member : dict -> key -> bool
#     (* insert dict key value -- Inserts a key-value pair into
#        dict. If the key is already present, updates the key to
#        have the new value. *)
```

```
#     val insert : dict -> key -> value -> dict
#     (* remove dict key -- Removes the key and its value from the
#        dictionary, if present. If the key is not present,
#        returns the original dictionary. *)
#     val remove : dict -> key -> dict
#   end ;;
module type DICT =
  sig
    type key
    type value
    type dict
    val empty : dict
    val lookup : dict -> key -> value option
    val member : dict -> key -> bool
    val insert : dict -> key -> value -> dict
    val remove : dict -> key -> dict
  end
```

We'll want a functor that builds dictionaries for all kinds of keys and values. In order to make sure we can compare the keys properly, including ordering them, we'll need a comparison function for keys as well. While we're at it, we might as well use a nicer convention for the comparison function, which will return a value of type

```
type order = Less | Equal | Greater ;;
```

The argument to the functor should thus satisfy the following signature:

```
# module type DICT_ARG =
#   sig
#     type key
#     type value
#     (* We need to reveal the order type so users of the
#        module can match against it to implement compare *)
#     type order = Less | Equal | Greater
#     (* Comparison function on keys compares two elements
#        and returns their order *)
#     val compare : key -> key -> order
#   end ;;
module type DICT_ARG =
  sig
    type key
    type value
    type order = Less | Equal | Greater
    val compare : key -> key -> order
  end
```

An implementation of such a functor is given here. It takes a module `D` satisfying `DICT_ARG`, providing all the needed information about the key and value types and the ordering of keys. It allows access to the key and value types via sharing constraints, so users of modules generated

by the functor can provide values of those types. This particular implementation of dictionaries is a simple list of key-value pairs, sorted by unique keys.

```
# module MakeOrderedDict (D : DICT_ARG)
#                        : (DICT with type key = D.key
#                                 and type value = D.value) =
#   struct
#     type key = D.key
#     type value = D.value
#
#     (* Invariant: sorted by key, no duplicate keys *)
#     type dict = (key * value) list
#
#     let empty = []
#
#     let rec lookup d k =
#       match d with
#       | [] -> None
#       | (k1, v1) :: d1 ->
#           let open D in
#           match compare k k1 with
#           | Equal -> Some v1
#           | Greater -> lookup d1 k
#           | Less -> None
#
#     let member d k =
#       match lookup d k with
#       | None -> false
#       | Some _ -> true
#
#     let rec insert d k v =
#       match d with
#       | [] -> [k, v]
#       | (k1, v1) :: d1 ->
#           let open D in
#           match compare k k1 with
#           | Less -> (k, v) :: d
#           | Equal -> (k, v) :: d1
#           | Greater -> (k1, v1) :: (insert d1 k v)
#
#     let rec remove d k =
#       match d with
#       | [] -> []
#       | (k1, v1) :: d1 ->
#           let open D in
#           match compare k k1 with
#           | Equal -> d1
#           | Greater -> (k1, v1) :: (remove d1 k)
#           | Less -> d
#   end ;;
module MakeOrderedDict :
  functor (D : DICT_ARG) ->
    sig
```

```
    type key = D.key
    type value = D.value
    type dict
    val empty : dict
    val lookup : dict -> key -> value option
    val member : dict -> key -> bool
    val insert : dict -> key -> value -> dict
    val remove : dict -> key -> dict
  end
```

A reverse index, recall, is just a dictionary for mapping `string` keys to string set values. (The latter we've already built as the type `StringSet.set`.) Let's build one using the `MakeOrderedDict` functor.

The argument to the functor should specify the key and value types and the ordering on keys:

```
# module StringStringSetDictArg
#        : (DICT_ARG with type key = string
#                     and type value = StringSet.set) =
#   struct
#     type key = string
#     type value = StringSet.set
#     type order = Less | Equal | Greater
#     let compare x y = if x < y then Less
#                       else if x = y then Equal
#                       else Greater
#   end ;;
module StringStringSetDictArg :
  sig
    type key = string
    type value = StringSet.set
    type order = Less | Equal | Greater
    val compare : key -> key -> order
  end
```

Now to generate an index module requires only a single line.

```
# module Index = MakeOrderedDict (StringStringSetDictArg) ;;
module Index :
  sig
    type key = StringStringSetDictArg.key
    type value = StringStringSetDictArg.value
    type dict = MakeOrderedDict(StringStringSetDictArg).dict
    val empty : dict
    val lookup : dict -> key -> value option
    val member : dict -> key -> bool
    val insert : dict -> key -> value -> dict
    val remove : dict -> key -> dict
  end
```

By making use of these generic constructs for sets and dictionaries, we can build a reverse index type easily, and implement query evaluation in a manner that is oblivious to, hence robust to any changes in,

the implementation of the sets and dictionaries. The code for `eval` can be as specified before, and repeated here.

```
# let rec eval (q : query)
#              (idx : Index.dict)
#            : StringSet.set =
#   match q with
#   | Word word -> (match Index.lookup idx word with
#                   | None -> StringSet.empty
#                   | Some v -> v)
#   | And (q1, q2) -> StringSet.intersection (eval q1 idx)
#                                            (eval q2 idx)
#   | Or (q1, q2) -> StringSet.union (eval q1 idx)
#                                    (eval q2 idx) ;;
val eval : query -> Index.dict -> StringSet.set = <fun>
```

More generally, modules allow separating an interface from its implementation, the key premise of abstract data types and modular programming, and OCaml's functors provide for constructing modules that operate generically.

12.7 Alternative methods for defining signatures and modules

We've already seen two ways to define a module subject to a particular signature. First is to name the signature explicitly using `module type`, and use that name in defining the module itself.

```
module type SIG_NAME =
  sig
    ...component declarations...
  end ;;

module ModuleName : SIG_NAME =
  struct
    ...component implementations...
  end ;;
```

Second is to place an unnamed signature directly constraining the module definition

```
module ModuleName : sig
         ...component declarations...
       end =
  struct
    ...component implementations...
  end ;;
```

useful on occasions where the signature is quite short and will only be used once, so retaining a name for it isn't needed.

There is a third method, widely used within OCaml's own implementation of library modules. All of the components defined in a `.ml`

file *automatically* constitute a module, whose name is generated by converting the first letter of the filename to uppercase. For example, if we have a file named `queue.ml` whose contents is

```
type 'a queue = 'a list
let empty_queue : 'a queue = []
let enqueue (elt : 'a) (q : 'a queue) : 'a queue =
  q @ [elt]
let dequeue (q : 'a queue) : 'a * 'a queue =
  match q with
  | [] -> raise (Invalid_argument
                   "dequeue: empty queue")
  | hd :: tl -> hd, tl
```

then we can refer in other files to `Queue.queue` to gain access to the type defined in that file, to `Queue.enqueue` to access the enqueueing function, and so forth. We can even place an `open Queue` at the top of another file to have unprefixed access to the components of the module.

How to define a signature for such a module though? OCaml looks for a file with the same prefix but the extension `.mli` (the `i` is for "interface"), which holds the component declarations for the signature. Thus, we should place in a file `queue.mli` these declarations:

```
type 'a queue
val empty_queue : 'a queue
val enqueue : 'a -> 'a queue -> 'a queue
val dequeue : 'a queue -> 'a * 'a queue
```

The `Queue` module will then be constrained by this signature, simply by virtue of the matching filenames.

12.7.1 Set and dictionary modules

The facilities for generating set modules – including the `SET` signature and `MakeOrderedSet` functor – might well be packaged up into a single module themselves. A file `set.ml` providing such a module might look like the following:

```
(* A Set Module *)

(*.....................................................
  Set interface
 *)

module type SET =
  sig
    type element (* elements of the set *)
    type set     (* sets formed from the elements *)

    (* The empty set *)
```

```
    val empty : set
    (* Returns true if set is empty; false otherwise *)
    val is_empty : set -> bool
    (* Adds element to existing set (if not already a member) *)
    val add : element -> set -> set
    (* Union of two sets *)
    val union : set -> set -> set
    (* Intersection of two sets *)
    val intersection : set -> set -> set
    (* Returns true iff element is in set *)
    val member : element -> set -> bool
  end ;;

(*..................................................
   An implementation for elements of ordered type
 *)

(* Module for types with a comparison function *)

module type COMPARABLE =
  sig
    (* The type of comparable elements *)
    type t
    (* We need to reveal the order type so users of the
       module can match against it *)
    type order = Less | Equal | Greater
    (* Comparison function compares two elements of the
       type and returns their order *)
    val compare : t -> t -> order
  end

(* Functor that generates sets for any comparable type *)

module MakeOrderedSet (Elements : COMPARABLE)
       : (SET with type element = Elements.t) =
  (* Implementation of SET as list of elements. Assumes
     list is sorted with no duplicates. *)
  struct
    type element = Elements.t
    type set = element list

    let empty = []
    let is_empty s = (s = [])
    let rec member elt s =
      match s with
      | [] -> false
      | hd :: tl ->
          let open Elements in
          (* so that Elements.compare, Elements.Less,
             etc. are in scope *)
          match compare elt hd with
          | Equal -> true
          | Less -> false
          | Greater -> member elt tl
```

```
    let rec add elt s =
      (* add the elt in the proper place in the
         ordered list *)
      match s with
      | [] -> [elt]
      | hd :: tl ->
          let open Elements in
          match compare elt hd with
          | Less -> elt :: s
          | Equal -> s
          | Greater -> hd :: add elt tl

    let union s1 s2 = List.fold_right add s1 s2

    let rec intersection xs ys =
      match xs, ys with
      | [], _ -> []
      | _, [] -> []
      | xh :: xt, yh :: yt ->
          let open Elements in
          match compare xh yh with
          | Equal -> xh :: intersection xt yt
          | Less -> intersection xt ys
          | Greater -> intersection xs yt
  end ;;
```

This file defines a module called `set` that enables usage like the following, to define and use a StringSet module:

```
module StringSet =
  let open Set in
  MakeOrderedSet
    (struct
       type t = string
       type order = Less | Equal | Greater
       let compare s t = if s < t then Less
                         else if s = t then Equal
                         else Greater
     end) ;;

let s = StringSet.create
        |> StringSet.add "a"
        |> StringSet.add "b"
        |> StringSet.add "a" ;;
```

12.8 Library Modules

Data structures like sets and dictionaries are so generally useful that you might think the language ought to provide them so that each individual programmer doesn't need to implement them. In fact, OCaml *does* provide these and many other data structures – as LIBRARY MODULES.

In particular, the `Set` library module provides functionality much like the `Set` module in the previous section, and the `Map` library module provides functionality much like our dictionary module and its `MakeOrderedDict` functor.

In later chapters, we'll happily avail ourselves of these built-in libraries. Nonetheless, it's still important to see how such simple and general abstract data structures can be provided as modules, for several reasons: to demystify what's going on in the library-provided modules, to instantiate the idea that the language itself is sufficient for implementing these ideas, and as examples to inspire ways to implement other, more application-specific abstract data structures.

12.9 Problem section: Image manipulation

We define here a signature for modules that deal with images and their manipulation.

```
module type IMAGING =
  sig
    (* types for images, which are composed of pixels *)
    type image
    type pixel
    (* an image size is a pair of ints giving number of
       rows and columns *)
    type size = int * int
    (* converting between integers and pixels *)
    val to_pixel : int -> pixel
    val from_pixel : pixel -> int
    (* apply an image filter, a function over pixels,
       to every pixel in an image *)
    val filter : (pixel -> pixel) -> image -> image
    (* apply an image filter to two images, combining
       the images pixel by pixel *)
    val filter2 : (pixel -> pixel -> pixel)
                  -> image -> image -> image
    (* return a "constant" image of the specified size
       where every pixel has the same value *)
    val const : pixel -> size -> image
    (* display the image in a graphics window *)
    val depict : image -> unit
  end ;;
```

The pixels that make up an image are specified by the following signature:

```
module type PIXEL =
  sig
    type t
    val to_pixel : int -> t
    val from_pixel : t -> int
  end
```

Problem 103

We'd like to implement a functor named `MakeImaging` for generating implementations of the `IMAGING` signature based on modules satisfying the `PIXEL` signature. How should such a functor start? Give the header line of such beginning with the keyword `module` and ending with the `= struct...`.

Here is a module implementing the `PIXEL` signature for integer pixels.

```
module IntPixel : (PIXEL with type t = int) =
  struct
    type t = int
    let to_pixel x = x
    let from_pixel x = x
  end ;;
```

Problem 104

Write code that uses the `IntPixel` module to define an imaging module called `IntImaging`.

Problem 105

Write code to use the `IntImaging` module that you defined in Problem 104 to display a 100 by 100 pixel image where all of the pixels have the constant integer value 5000.

12.10 Problem section: An abstract data type for intervals

A good candidate for an abstract data type is the INTERVAL. Abstractly speaking, an interval is a region between two points, where all that is required of points is that we be able to compare them as an ordering (so that we have a well-defined notion of "between"). That is, points ought to obey the following signature, which may look familiar, as you've seen it in other contexts:

```
module type COMPARABLE =
  sig
    type t
    type order = Less | Equal | Greater
    val compare : t -> t -> order
  end ;;
```

Intervals come up in many different contexts. As an informal example, calendars need to associate events with time intervals, such as 3-4pm or 11:30am-3:30pm; the endpoints in this case would be times. Natural operations over intervals include: the construction of an interval between two points, the extraction of the endpoints of an interval, taking the union of two intervals (the smallest interval containing both) or their intersection, and determining the relation between two intervals (whether they are disjoint, overlapping, or one contains the other). Here is a signature that provides for this functionality.

```
module type INTERVAL =
  sig
```

```
  type point
  type interval
  type relation = Disjoint | Overlaps | Contains
  (* Returns the interval between two points *)
  val interval : point -> point -> interval
  (* Returns the endpoints of an interval as a pair
     with the first point less than the second. *)
  val endpoints : interval -> point * point
  (* Returns the union of two intervals *)
  val union : interval -> interval -> interval
  (* Returns the relation holding between two intervals *)
  val relation : interval -> interval -> relation
end ;;
```

The possible relations between two intervals are depicted in Figure 12.3. (For the interval arithmetic cognoscenti, we've left out many details, such as whether intervals are open or closed; more fine-grained relations; and many other useful operations on intervals. These issues are beyond the scope of this problem.)

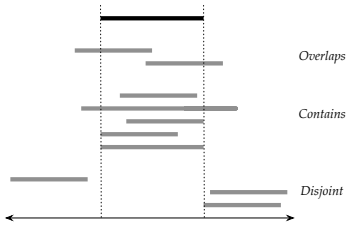

Figure 12.3: A diagrammatic depiction of the possible relations holding between two intervals. In the diagram, the gray intervals in the three groups below the black interval are in the "overlaps" (top 2), "contains" (next 5), and "disjoint" (bottom 3) relations, respectively, with the black interval at top. The vertical dotted lines depict the endpoints of the black interval.

Problem 106

We'd like to have a functor named `MakeInterval` for generating implementations of the `INTERVAL` signature based on modules satisfying the `COMPARABLE` signature. How should such a functor start? Give the header line of such a functor definition beginning with the keyword `module` and ending with the `= struct...`.

Problem 107

An appropriate module satisfying `COMPARABLE` for the purpose of generating *discrete* time intervals would be one where the type is `int`, with an appropriate comparison function. Define a module named `DiscreteTime` satisfying `COMPARABLE` where the type is `int`. Make sure the type is accessible outside the module.

Problem 108

Now use the functor `MakeInterval` to define a module `DiscreteTimeInterval` that provides interval functionality over discrete times as defined by the module `DiscreteTime` above.

Problem 109

The intersection of two intervals is only well-defined if the intervals are not disjoint. Assume that the `DiscreteTimeInterval` module has been opened, allowing you to make use of everything in its signature. Now, define a function `intersection : interval -> interval -> interval option` that takes two intervals and returns `None` if they are disjoint and otherwise returns their intersection (embedded appropriately in the option type).

Problem 110

Provide three different unit tests that would be useful in testing the correctness of the `DiscreteTimeInterval` module.

12.11 Problem section: Mobiles

The artist Alexander Calder (1898-1976) is well known for his distinctive mobiles, sculptures with different shaped objects hung from a cascade of connecting metal bars. An example is given in Figure 12.4.

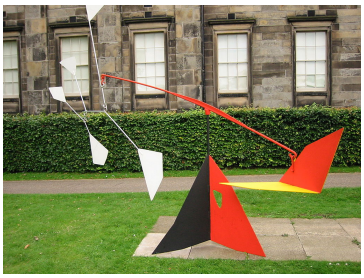

Figure 12.4: Alexander Calder's *L'empennage* (1953).

His mobiles are made with varying shapes at the ends of the connectors – circles, ovals, fins. The exquisite balance of the mobiles

depends on the weights of the various components. In the next few exercises of this problem, you will model the structure of mobiles as binary trees such that one can determine if a Calder-like mobile design is balanced or not. Let's start with the objects at the ends of the connectors. For our purposes, the important properties of an object will be its shape and its weight (in arbitrary units; you can interpret them as pounds).

Problem 111

Define a `weight` type consisting of a single floating point weight.

Problem 112

Define a `shape` type, a variant type that allows for three different shapes: circles, ovals, and fins.

Problem 113

Define an `object` type that will be used to store information about the objects at the ends of the connectors, in particular, their weight and their shape.

A mobile can be modeled as a kind of binary tree, where the leaves of the tree, representing the objects, are elements of type `obj`, and the internal nodes, representing the connectors, have a weight, and each internal node (connector) connects two submobiles. Rather than directly writing code for a `mobile` type, though, we'll digress to build a more general binary tree module, and then model mobiles using that. An appropriate signature `BINTREE` for a simple binary tree module might be the following:

```
module type BINTREE =
  sig
    type leaft (* the type for the leaves of the tree *)
    type nodet (* the type for the internal nodes of the tree *)
    type tree  (* the type for the trees themselves *)

    val make_leaf : leaft -> tree
    val make_node : nodet -> tree -> tree -> tree
    val walk : (leaft -> 'a)
                -> (nodet -> 'a -> 'a -> 'a) -> tree -> 'a
  end ;;
```

This module signature specifies separate types for the leaves of trees and the internal nodes of trees, along with a type for the trees themselves; functions for constructing leaf and node trees; and a single function to "walk" the tree. (We'll come back to the `walk` function later.) In addition to the signature for binary tree modules, we would need a way of generating implementations of modules satisfying the `BINTREE` signature, which we'll do with a functor `MakeBintree`. The `MakeBinTree` functor takes an argument module of type `BINTREE_ARG` that packages up the particular types for the leaves and nodes, that is, the types to use for `leaft` and `nodet`. The following module signature will work:

```
module type BINTREE_ARG =
  sig
    type leaft
    type nodet
  end ;;
```

Problem 114

Write down the header of a definition of a functor named `MakeBintree` taking a `BINTREE_ARG` argument, which generates modules satisfying the `BINTREE` signature. Keep in mind the need for users of the functor-generated modules to access appropriate aspects of the generated trees. (You don't need to fill in the actual implementation of the functor.)

Using the `MakeBintree` functor described above, you can now generate a `Mobile` module, which has `objs` at the leaves and `weights` at the interior nodes.

Problem 115

Define a module `Mobile` using the functor `MakeBintree`.

Problem 116

You've just used the `MakeBintree` functor without ever seeing its implementation. Why is this possible?

You can now build a representation of a mobile using the functions that the `Mobile` module makes available.

Problem 117

Define a value `mobile1` of type `Mobile.tree` that represents a mobile structured as the one depicted in Figure 12.5.

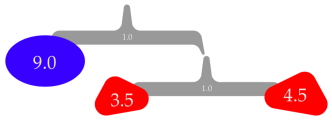

Figure 12.5: A simple Calder-style mobile. The depicted mobile has two connectors and three objects (an oval and two fins). The connectors each weigh 1.0, and the objects' weights are as given in the figure.

The `walk` function, of type `(leaft -> 'a) -> (nodet -> 'a -> 'a -> 'a) -> tree -> 'a`, is of special interest, since it is the sole method for performing computations over these binary trees. The function is a kind of fold that works over trees instead of lists. It takes two functions – one for leaves and one for nodes – and applies these functions to a tree to generate a single value. The leaf function takes a `leaft` and returns some value of type `'a`. The node function takes a `nodet`, as well as the two `'a` values recursively returned by walking its two subtrees, and computes the value for the node itself. For example, we can use `walk` to define a function size that counts how many objects there are in a mobile. The function uses the fact that leaves are of size `1` and the size of a non-leaf is the sum of the sizes of its subtrees.

```
let size mobile =
  Mobile.walk (fun _leaf -> 1)
              (fun _node left_size right_size ->
                 left_size + right_size)
              mobile ;;
```

Problem 118

What is the type of `size`?

Problem 119

Use the fact that the `walk` function is curried to give a slightly more concise definition for size.

Problem 120

Use the `walk` function to implement a function `shape_count : shape -> Mobile.tree -> int` that takes a shape and a mobile (in that order), and returns the number of objects in the mobile that have that particular shape.

A mobile is said to be *balanced* if every connector has the property that the total weight of all components (that is, objects and connectors) of its left submobile is the same as the total weight of all components of its right submobile. (In actuality, we'd have to worry about other things like the relative lengths of the arms of the connectors, but we'll ignore all that.)

Problem 121

Is the mobile shown balanced? Why or why not?

Problem 122

Implement a function `balance : Mobile.tree -> weight option` that takes a mobile, and returns `None` if the argument mobile is not balanced, and `Some w` if the mobile is balanced, where `w` is the total weight of the mobile.

12.12 Supplementary material

- Lab 7: Modules and abstract data types
- Lab 8: Functors
- Problem set A.5: Ordered collections