Part III

Application Structure

The third part of this work is dedicated to application development and describes two ways of organizing applications: modules and objects. The goal is to easily structure an application for incremental and rapid development, maintenance facilitated by the ability to change gracefully, and the possibility of reusing large parts for future development.

We have already presented the language's predefined modules (see chapter 8) viewed as compilation units. Objective Caml's module language supports on the one hand the definition of new simple modules in order to build one's own libraries, perhaps including abstract types, and on the other hand the definition of modules parameterized by other modules, called *functors*. The advantage of this parameterization lies in being able to "apply" a module to different argument modules in order to create specialized modules. Communication between modules is thus explicit, via the parameter module signature, which contains the types of its global declarations. However, nothing stops you from applying a functor to a module with a more extended signature, as long as it remains compatible with the specified parameter signature.

Besides, the Objective Caml language has an object-oriented extension. First of all object-oriented programming permits structured communication between objects. Rather than applying a function to some arguments, one sends a message (a request) to an object which knows how to deal with it. The object, an instance of a class (a structure gathering together data and methods), then executes the corresponding code. The main relation between classes is inheritance, which lets one describe subclasses which retain all the declarations of the ancestor class. Late binding between the name of a message and the corresponding code within the object takes place during program execution. Nevertheless Objective Caml typing guarantees that the receiving object will always have a method of this name, otherwise type inference would have raised a compile-time error. The second important relation is subtyping, where an object of a certain class can always be used in place of an object of another class. In this way a new type of polymorphism is introduced: inclusion polymorphism.

Finally the construction of a graphical interface, begun in chapter 5, uses different event management models. One puts together in an interface several components with respect to which the user or the system can produce events. The association of a component with a handler for one or more events taking place on it allows one to easily add to and modify such interfaces. The component-event-handler association can be cloaked in several forms: definition of a function (called a callback), inheritance with redefinition of handler methods, or finally registration of a handling object (delegation model).

Chapter 14 is a presentation of modular programming. The different prevailing terminologies of abstract data types and module languages are explained and illustrated by simple modules. Then the module language is detailed. The correspondence between modules (simple or not) and compilation units is made clear.

Chapter 15 contains an introduction to object-oriented programming. It brings a new way of structuring Objective Caml programs, an alternative to modules. This chapters shows how the notions of object-oriented programming (simple and multiple inheritance, abstract classes, parameterized classes, late binding) are articulated with respect to the language's type system, and extend it by the subtyping relation to inclusion polymorphism.

Chapter 16 compares the two preceding software models and explains what factors to consider in deciding between the two, while also demonstrating how to simulate one by the other. It treats various cases of mixed models. Mixing leads to the enrichment of each of these two models, in particular with parameterized classes using the abstract type of a module.

Chapter 17 presents two classes of applications: two-player games, and the construction of a world of virtual robots. The first example is organized via various parameterized modules. In particular, a parameterized module is used to represent games for application of the minimax $\alpha\beta$ algorithm. It is then applied to two specific games: Connect 4 and Stone Henge. The second example uses an object model of a world and of abstract robots, from which, by inheritence, various simulations are derived. This example is presented in chapter 21.

14

Programming with Modules

Modular design and modular programming support the decomposition of a program into several *software units*, also called *modules*, which can be developed largely independently. A module can be compiled separately from the other modules comprising the program. Consequently, the developer of a program that uses a module does not need access to the source code of the module: the compiled code of the module is enough for building an executable program. However, the programmer must know the *interface* of the modules used, that is, which values, functions, types, exceptions, or even sub-modules are provided by the module, under which names, and with which types.

Explicitly writing down the interface of a module hides the details of its implementation from the programs that use this module. All these programs know about the module are the names and types of exported definitions; their exact implementations are not known. Thus, the maintainer of the module has considerable flexibility in evolving the module implementation: as long as the interface is unchanged and the semantics are preserved, users of the module will not notice the change in implementation. This can greatly facilitate the maintenance and evolution of large programs. Like local declarations, a module interface also supports hiding parts of the implementation that the module designer does not wish to publicize. An important application of this hiding mechanism is the implementation of *abstract data types.*

Finally, advanced module systems such as that of Objective Caml support the definition of *parameterized modules*, also called *generics.* These are modules that take other modules as parameters, thus increasing opportunities for code reuse.

Chapter Outline

Section 1 illustrates Objective Caml modules on the example of the `Stack` module from the standard library, and develops an alternate implementation of this module with the same interface. Section 2 introduces the *module language* of Objective Caml in the case of simple modules, and shows some of its uses. In particular, we discuss type sharing between modules. Section 3 covers parameterized modules, which are called *functors* in Objective Caml. Finally, section 4 develops an extended example of modular programming: managing bank accounts with multiple views (the bank, the customer) and several parameters.

Modules as Compilation Units

The Objective Caml distribution includes a number of predefined modules. We saw in chapter 8 how to use these modules in a program. Here, we will show how users can define similar modules.

Interface and Implementation

The module `Stack` from the distribution provides the main functions on stacks, that is, queues with "last in, first out" discipline.

```
# let queue = Stack.create () ;;
val queue : '_a Stack.t = <abstr>
# Stack.push 1 queue ; Stack.push 2 queue ; Stack.push 3 queue ;;
- : unit = ()
# Stack.iter (fun n → Printf.printf "%d " n)  queue ;;
3 2 1 - : unit = ()
```

Since Objective Caml is distributed with full source code, we can look at the actual implementation of stacks.

```
ocaml-2.04/stdlib/stack.ml
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with  hd::tl → s.c <- tl; hd  |  []  → raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

We see that the type of stacks (written `Stack.t` outside the `Stack` module and just `t` inside) is a record with one mutable field containing a list. The list holds the contents of the stack, with the list head corresponding to the stack top. Stack operations are implemented as the basic list operations applied to the field of the record.

Armed with this insider's knowledge, we could try to access directly the list representing a stack. However, Objective Caml will not let us do this.

```
# let list = queue.c ;;
Characters 12-19:
Unbound label c
```

The compiler complains as if it did not know that `Stack.t` is a record type with a field `c`. It is actually the case, as we can see by looking at the interface of the `Stack` module.

ocaml-2.04/stdlib/stack.mli

```
(* Module [Stack]: last-in first-out stacks *)
(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t         (* The type of stacks containing elements of type ['a]. *)

exception Empty   (* Raised when [pop] is applied to an empty stack. *)

val create: unit → 'a t
        (* Return a new stack, initially empty. *)
val push: 'a → 'a t → unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t → 'a
        (* [pop s] removes and returns the topmost element in stack [s],
           or raises [Empty] if the stack is empty. *)
val clear : 'a t → unit
        (* Discard all elements from a stack. *)
val length: 'a t → int
        (* Return the number of elements in a stack. *)
val iter: ('a → unit) → 'a t → unit
        (* [iter f s] applies [f] in turn to all elements of [s],
           from the element at the top of the stack to the element at the
           bottom of the stack. The stack itself is unchanged. *)
```

In addition to comments documenting the functions of the module, this file lists explicitly the value, type and exception identifiers defined in the file `stack.ml` that should be visible to clients of the `Stack` module. More precisely, the interface declares the names and type specifications for these exported definitions. In particular, the type name `t` is exported, but the representation of this type (that is, as a record with one `c` field) is not given in this interface. Thus, clients of the `Stack` module do not know how the type `Stack.t` is represented, and cannot access directly values of this type. We say that the type *Stack.t* is *abstract*, or *opaque*.

The interface also declares the functions operating on stacks, giving their names and types. (The types must be provided explicitly so that the type checker can check that these functions are correctly used.) Declaration of values and functions in an interface is achieved via the following construct:

Syntax : $\boxed{\texttt{val } nom \texttt{ : } type}$

Relating Interfaces and Implementations

As shown above, the `Stack` is composed of two parts: an implementation providing definitions, and an interface providing declarations for those definitions that are exported. All module components declared in the interface must have a matching definition in the implementation. Also, the types of values and functions as defined in the implementation must match the types declared in the interface.

The relationship between interface and implementation is not symmetrical. The implementation can contain more definitions than requested by the interface. Typically, the definition of an exported function can use auxiliary functions whose names will not appear in the interface. Such auxiliary functions cannot be called directly by a client of the module. Similarly, the interface can restrict the type of a definition. Consider a module defining the function `id` as the identity function (`let id x = x`). Its interface can declare `id` with the type `int -> int` (instead of the more general `'a -> 'a`). Then, clients of this module can only apply `id` to integers.

Since the interface of a module is clearly separated from its implementation, it becomes possible to have several implementations for the same interface, for instance to test different algorithms or data structures for the same operations. As an example, here is an alternate implementation for the `Stack` module, based on arrays instead of lists.

```
type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [||] }
let clear s = s.sp <- 0; s.c <- [||]
let size = 5
let increase s = s.c <- Array.append s.c (Array.create size s.c.(0))

let push x s =
  if s.sp >= Array.length s.c then increase s ;
  s.c.(s.sp) <- x ;
  s.sp <- succ s.sp

let pop s =
  if s.sp = 0 then raise Empty
  else let x = s.c.(s.sp) in s.sp <- pred s.sp ; x

let length s = s.sp
let iter f s = for i = pred s.sp downto 0 do f s.sc.(i) done
```

This new implementation satisfies the requisites of the interface file `stack.mli`. Thus, it can be used instead of the predefined implementation of `Stack` in any program.

Separate Compilation

Like most modern programming languages, Objective Caml supports the decomposition of programs into multiple compilation units, separately compiled. A compilation unit is composed of two files, an implementation file (with extension `.ml`) and an interface file (with extension `.mli`). Each compilation unit is viewed as a module. Compiling the implementation file `name.ml` defines the module named `Name`[1].

Values, types and exceptions defined in a module can be referenced either via the *dot notation* (`Module.identifier`), also known as *qualified identifiers*, or via the **`open`** construct.

a.ml	b.ml
type *t* = { *x*:*int* ; *y*:*int* } ;;	**let val** = { *A.x* = 1 ; *A.y* = 2 } ;;
let *f c* = *c.x* + *c.y* ;;	*A.f* **val** ;;
	open *A* ;;
	f **val** ;;

An interface file (`.mli` file) must be compiled using the `ocamlc -c` command before any module that depends on this interface is compiled; this includes both clients of the module and the implementation file for this module as well.

If no interface file is provided for an implementation file, Objective Caml considers that the module exports everything; that is, all identifiers defined in the implementation file are present in the implicit interface with their most general types.

The linking phase to produce an executable file is performed as described in chapter 7: the `ocamlc` command (without the `-c` option), followed by the object files for all compilation units comprising the program. Warning: object files must be provided on the command line in dependency order. That is, if a module `B` references another module `A`, the object file `a.cmo` must precede `b.cmo` on the linker command line. Consequently, cross dependencies between two modules are forbidden.

For instance, to generate an executable file from the source files `a.ml` and `b.ml`, with matching interface files `a.mli` and `b.mli`, we issue the following commands:

```
> ocamlc -c a.mli
> ocamlc -c a.ml
> ocamlc -c b.mli
> ocamlc -c b.ml
> ocamlc a.cmo b.cmo
```

Compilation units, composed of one interface file and one implementation file, support separate compilation and information hiding. However, their abilities as a general program structuring tool are low. In particular, there is a one-to-one connection

1. Both files `name.ml` and `Name.ml` result in the same module name.

between modules and files, preventing a program to use simultaneously several implementations of a given interface, or also several interfaces for the same implementation. Nested modules and module parameterization are not supported either. To palliate those weaknesses, Objective Caml offers a module language, with special syntax and linguistic constructs, to manipulate modules inside the language itself. The remainder of this chapter introduces this module language.

The Module Language

The Objective Caml language features a sub-language for modules, which comes in addition to the core language that we have seen so far. In this module language, the interface of a module is called a *signature* and its implementation is called a *structure*. When there is no ambiguity, we will often use the word "module" to refer to a structure.

The syntax for declaring signatures and structures is as follows:

Syntax :

```
module type NAME =
  sig
    interface declarations
  end
```

Syntax :

```
module Name =
  struct
    implementation definitions
  end
```

Warning The name of a module *must* start with an uppercase letter. There are no such case restrictions on names of signatures, but by convention we will use names in uppercase for signatures.

Signatures and structures do not need to be bound to names: we can also use anonymous signature and structure expressions, writing simply

Syntax : `sig declarations end`

Syntax : `struct definitions end`

We write *signature* and *structure* to refer to either names of signatures and structures, or anonymous signature and structure expressions.

Every structure possesses a default signature, computed by the type inference system, which reveals all the definitions contained in the structure, with their most general types. When defining a structure, we can also indicate the desired signature by adding a signature constraint (similar to the type constraints from the core language), using one of the following two syntactic forms:

Syntax : $\boxed{\texttt{module } \textit{Name} \texttt{ : } \textit{signature} \texttt{ = } \textit{structure}}$

Syntax : $\boxed{\texttt{module } \textit{Name} \texttt{ = (} \textit{structure} \texttt{ : } \textit{signature}\texttt{)}}$

When an explicit signature is provided, the system checks that all the components declared in the signature are defined in the structure *structure*, and that the types are consistent. In other terms, the system checks that the explicit signature provided is "included in", or implied by, the default signature. If so, *Name* is viewed in the remainder of the code with the signature "*signature*", and only the components declared in the signature are accessible to the clients of the module. (This is the same behavior we saw previously with interface files.)

Access to the components of a module is via the dot notation:

Syntax : $\boxed{\textit{Name}_1\texttt{.}\textit{name}_2}$

We say that the name $name_2$ is *qualified* by the name $Name_1$ of its defining module.

The module name and the dot can be omitted using a directive to *open* the module:

Syntax : $\boxed{\texttt{open } \textit{Name}}$

In the scope of this directive, we can use short names $name_2$ to refer to the components of the module *Name*. In case of name conflicts, opening a module hides previously defined entities with the same names, as in the case of identifier redefinitions.

Two Stack Modules

We continue the example of stacks by recasting it in the module language. The signature for a stack module is obtained by wrapping the declarations from the `stack.mli` file in a signature declaration:

```
# module type STACK =
    sig
      type 'a t
      exception Empty
      val create: unit → 'a t
      val push: 'a → 'a t → unit
      val pop: 'a t → 'a
      val clear : 'a t → unit
      val length: 'a t → int
      val iter: ('a → unit) → 'a t → unit
    end ;;
module type STACK =
  sig
    type 'a t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
```

A first implementation of stacks is obtained by reusing the `Stack` module from the standard library:

```
# module StandardStack = Stack ;;
module StandardStack :
  sig
    type 'a t = 'a Stack.t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
```

We then define an alternate implementation based on arrays:

```
# module MyStack =
    struct
      type 'a t = { mutable sp : int; mutable c : 'a array }
      exception Empty
      let create () = { sp=0 ; c = [||] }
      let clear s = s.sp <- 0; s.c <- [||]
      let increase s x =  s.c <- Array.append s.c (Array.create 5 x)
      let push x s =
        if s.sp >= Array.length s.c then increase s x;
        s.c.(s.sp) <- x;
        s.sp <- succ s.sp
      let pop s =
        if s.sp =0 then raise Empty
        else (s.sp <- pred s.sp ; s.c.(s.sp))
      let length s = s.sp
      let iter f s = for i = pred s.sp downto 0 do f s.c.(i) done
    end ;;
module MyStack :
  sig
    type 'a t = { mutable sp: int; mutable c: 'a array }
    exception Empty
    val create : unit -> 'a t
    val clear : 'a t -> unit
    val increase : 'a t -> 'a -> unit
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val length : 'a t -> int
    val iter : ('a -> 'b) -> 'a t -> unit
  end
```

These two modules implement the type `t` of stacks by different data types.

```
# StandardStack.create () ;;
- : '_a StandardStack.t = <abstr>
# MyStack.create () ;;
- : '_a MyStack.t = {MyStack.sp=0; MyStack.c=[||]}
```

To abstract over the type representation in `Mystack`, we add a signature constraint by the `STACK` signature.

```
# module MyStack = (MyStack : STACK) ;;
module MyStack : STACK
# MyStack.create() ;;
- : '_a MyStack.t = <abstr>
```

The two modules `StandardStack` and `MyStack` implement the same interface, that is, provide the same set of operations over stacks, but their `t` types are different. It is therefore impossible to apply operations from one module to values from the other module:

```
# let s = StandardStack.create() ;;
val s : '_a StandardStack.t = <abstr>
# MyStack.push 0 s ;;
Characters 15-16:
This expression has type 'a StandardStack.t = 'a Stack.t
but is here used with type int MyStack.t
```

Even if both modules implemented the *t* type by the same concrete type, constraining `MyStack` by the signature `STACK` suffices to abstract over the *t* type, rendering it incompatible with any other type in the system and preventing sharing of values and operations between the various stack modules.

```
# module S1 = ( MyStack : STACK ) ;;
module S1 : STACK
# module S2 = ( MyStack : STACK ) ;;
module S2 : STACK
# let s = S1.create () ;;
val s : '_a S1.t = <abstr>
# S2.push 0 s ;;
Characters 10-11:
This expression has type 'a S1.t but is here used with type int S2.t
```

The Objective Caml system compares abstract types by names. Here, the two types `S1.t` and `S2.t` are both abstract, and have different names, hence they are considered as incompatible. It is precisely this restriction that makes type abstraction effective, by preventing any access to the definition of the type being abstracted.

Modules and Information Hiding

This section shows additional examples of signature constraints hiding or abstracting definitions of structure components.

Hiding Type Implementations

Abstracting over a type ensures that the only way to construct values of this type is via the functions exported from its definition module. This can be used to restrict the values that can belong to this type. In the following example, we implement an abstract type of integers which, by construction, can never take the value 0.

```
# module Int_Star =
    ( struct
        type t = int
        exception Isnul
        let of_int = function 0 → raise Isnul | n → n
        let mult = ( * )
      end
    :
      sig
        type t
        exception Isnul
        val of_int : int → t
        val mult : t → t → t
      end
    ) ;;
module Int_Star :
  sig type t exception Isnul val of_int : int -> t val mult : t -> t -> t end
```

Hiding Values

We now define a symbol generator, similar to that of page 103, using a signature constraint to hide the state of the generator.

We first define the signature GENSYM exporting only two functions for generating symbols.

```
# module type GENSYM =
    sig
      val reset : unit → unit
      val next : string → string
    end ;;
```

We then implement this signature as follows:

```
# module Gensym : GENSYM =
    struct
      let c = ref 0
      let reset () = c:=0
      let next s = incr c ; s ^ (string_of_int !c)
    end;;
module Gensym : GENSYM
```

The reference `c` holding the state of the generator `Gensym` is not accessible outside the two exported functions.

```
# Gensym.reset();;
- : unit = ()
# Gensym.next "T";;
- : string = "T1"
# Gensym.next "X";;
- : string = "X2"
# Gensym.reset();;
- : unit = ()
# Gensym.next "U";;
- : string = "U1"
# Gensym.c;;
Characters 0-8:
Unbound value Gensym.c
```

The definition of `c` is essentially local to the structure `Gensym`, since it is hidden by the associated signature. The signature constraint achieves more simply the same goal as the local definition of a reference in the definition of the two functions `reset_s` and `new_s` on page 103.

Multiple Views of a Module

The module language and its signature constraints support taking several views of a given structure. For instance, we can have a "super-user interface" for the module `Gensym`, allowing the symbol counter to be reset, and a "normal user interface" that permits only the generation of new symbols, but no other intervention on the counter. To implement the latter interface, it suffices to declare the signature:

```
# module type USER_GENSYM =
    sig
      val next : string → string
    end;;
module type USER_GENSYM = sig val next : string -> string end
```

We then implement it by a mere signature constraint.

```
# module UserGensym = (Gensym : USER_GENSYM) ;;
module UserGensym : USER_GENSYM
# UserGensym.next "U" ;;
- : string = "U2"
# UserGensym.reset() ;;
Characters 0-16:
Unbound value UserGensym.reset
```

The `UserGensym` module fully reuses the code of the `Gensym` module. In addition, both modules share the same counter:

```
# Gensym.next "U" ;;
- : string = "U3"
# Gensym.reset() ;;
- : unit = ()
# UserGensym.next "V" ;;
- : string = "V1"
```

Type Sharing between Modules

As we saw on page 411, abstract types with different names are incompatible. This can be problematic when we wish to share an abstract type between several modules. There are two ways to achieve this sharing: one is via a special sharing construct in the module language; the other one uses the lexical scoping of modules.

Sharing via Constraints

The following example illustrates the sharing issue. We define a module `M` providing an abstract type *M.t*. We then restrict `M` on two different signatures exporting different subsets of operations.

```
# module M =
  (
    struct
      type t = int ref
      let create() = ref 0
      let add x = incr x
      let get x = if !x>0 then (decr x; 1) else failwith "Empty"
    end
    :
    sig
      type t
      val create : unit → t
      val add : t → unit
      val get : t → int
    end
  ) ;;

# module type S1 =
    sig
      type t
      val create : unit → t
      val add : t → unit
    end ;;

# module type S2 =
    sig
      type t
      val get : t → int
    end ;;
# module M1 = (M:S1) ;;
module M1 : S1
# module M2 = (M:S2) ;;
module M2 : S2
```

As written above, the types *M1.t* and *M2.t* are incompatible. However, we would like to say that both types are abstract but identical. To do this, Objective Caml offers special syntax to declare a type equality over an abstract type in a signature.

Syntax : *NAME* **with type** t_1 **=** t_2 **and** ...

This type constraint forces the type t_1 declared in the signature *NAME* to be equal to the type t_2.

Type constraints over all types exported by a sub-module can be declared in one operation with the syntax

Syntax : *NAME* **with module** $Name_1$ **=** $Name_2$

Using these type sharing constraints, we can declare that the two modules `M1` and `M2` define identical abstract types.

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t val create : unit -> t val add : t -> unit end
# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig type t = M.t val get : t -> int end
# let x = M1.create() in M1.add x ;  M2.get x ;;
- : int = 1
```

Sharing and Nested Modules

Another possibility for ensuring type sharing is to use nested modules. We define two sub-modules (`M1` et `M2`) sharing an abstract type defined in the enclosing module `M`.

```
# module M =
    ( struct
        type t = int ref
        module M_hide =
          struct
            let create() = ref 0
            let add x = incr x
            let get x = if !x>0 then (decr x; 1) else failwith "Empty"
          end
        module M1 = M_hide
        module M2 = M_hide
      end
     :
      sig
        type t
        module M1 : sig  val create : unit → t  val add : t → unit end
        module M2 : sig  val get : t → int  end
      end ) ;;
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

As desired, values created by `M1` can be operated upon by `M2`, while hiding the representation of these values.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>
# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

This solution is heavier than the previous solution based on type sharing constraints: the functions from `M1` and `M2` can only be accessed via the enclosing module `M`.

Extending Simple Modules

Modules are closed entities, defined once and for all. In particular, once an abstract type is defined using the module language, it is impossible to add further operations on the abstract type that depend on the type representation without modifying the module definition itself. (Operations derived from existing operations can of course be added later, outside the module.) As an extreme example, if the module exports no creation function, clients of the module will never be able to create values of the abstract type!

Therefore, adding new operations that depend on the type representation requires editing the sources of the module and adding the desired operations in its signature and structure. Of course, we then get a different module, and clients need to be recompiled. However, if the modifications performed on the module signature did not affect the components of the original signature, the remainder of the program remains correct and does not need to be modified, just recompiled.

Parameterized Modules

Parameterized modules are to modules what functions are to base values. Just like a function returns a new value from the values of its parameters, a parameterized module builds a new module from the modules given as parameters. Parameterized modules are also called *functors.*

The addition of functors to the module language increases the opportunities for code reuse in structures.

Functors are defined using a function-like syntax:

Syntax : **functor (** *Name* **:** *signature* **)** -> *structure*

```
# module Couple = functor ( Q : sig type t end ) →
    struct type couple = Q.t * Q.t end ;;
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

As for functions, syntactic sugar is provided for defining and naming a functor:

Syntax : **module** $Name_1$ **(** $Name_2$ **:** *signature* **) =** *structure*

```
# module Couple ( Q : sig type t end ) =  struct type couple = Q.t * Q.t end ;;
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

A functor can take several parameters:

Syntax :
functor ($Name_1$ **:** $signature_1$ **)** ->
  ⋮
functor ($Name_n$ **:** $signature_n$ **)** ->
  structure

The syntactic sugar for defining and naming a functor extends to multiple-argument functors:

Syntax : **module** *Name* **(**$Name_1$ **:** $signature_1$ **)** … **(** $Name_n$ **:** $signature_n$ **) =**
  structure

The application of a functor to its arguments is written thus:

Syntax : **module** *Name* **=** *functor* **(** $structure_1$ **)** … **(** $structure_n$ **)**

Note that each parameter is written between parentheses. The result of the application can be either a simple module or a partially applied functor, depending on the number of parameters of the functor.

Warning There is no equivalent to functors at the level of signature: it is not possible to build a signature by application of a “functorial signature” to other signatures.

A closed functor is a functor that does not reference any module except its parameters. Such a closed functor makes its communications with other modules entirely explicit. This provides maximal reusability, since the modules it references are determined at application time only. There is a strong parallel between a closed function (without free variables) and a closed functor.

Functors and Code Reuse

The Objective Caml standard library provides three modules defining functors. Two of them take as argument a module implementing a totally ordered data type, that is, a module with the following signature:

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

Function `compare` takes two arguments of type `t` and returns a negative integer if the first is less than the second, zero if both are equal, and a positive integer if the first is greater than the second. Here is an example of totally ordered type: pairs of integers equipped with lexicographic ordering.

```
# module OrderedIntPair =
    struct
      type t = int * int
      let compare (x1,x2) (y1,y2) =
        if x1 < y1 then -1
        else if x1 > y1 then 1
        else if x2 < y2 then -1
        else if x2 > y2 then 1
        else 0
    end ;;
module OrderedIntPair :
  sig type t = int * int val compare : 'a * 'b -> 'a * 'b -> int end
```

The functor `Make` from module `Map` returns a module that implements association tables whose keys are values of the ordered type passed as argument. This module provides operations similar to the operations on association lists from module `List`, but using a more efficient and more complex data structure (balanced binary trees).

```
# module AssocIntPair = Map.Make (OrderedIntPair) ;;
module AssocIntPair :
  sig
    type key = OrderedIntPair.t
    and 'a t = 'a Map.Make(OrderedIntPair).t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val remove : key -> 'a t -> 'a t
    val mem : key -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val map : ('a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  end
```

The `Make` functor allows to construct association tables over any key type for which we can write a `compare` function.

The standard library module `Set` also provides a functor named `Make` taking an ordered type as argument and returning a module implementing sets of sets of values of this type.

```
# module SetIntPair = Set.Make (OrderedIntPair) ;;
module SetIntPair :
  sig
    type elt = OrderedIntPair.t
    and t = Set.Make(OrderedIntPair).t
    val empty : t
    val is_empty : t -> bool
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val singleton : elt -> t
    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
  end
```

The type `SetIntPair.t` is the type of sets of integer pairs, with all the usual set operations provided in `SetIntPair`, including a set comparison function `SetIntPair.-compare`. To illustrate the code reuse made possible by functors, we now build sets of sets of integer pairs.

```
# module SetofSet = Set.Make (SetIntPair) ;;

# let x = SetIntPair.singleton (1,2) ;;           (* x = { (1,2) }        *)
val x : SetIntPair.t = <abstr>
# let y = SetofSet.singleton SetIntPair.empty ;;  (* y = { {} }           *)
val y : SetofSet.t = <abstr>
# let z = SetofSet.add x y ;;                     (* z = { {(1,2)} ; {} } *)
val z : SetofSet.t = <abstr>
```

The `Make` functor from module `Hashtbl` is similar to that from the `Map` module, but implements (imperative) hash tables instead of (purely functional) balanced trees. The argument to `Hashtbl.Make` is slightly different: in addition to the type of the keys for the hash table, it must provide an equality function testing the equality of two keys (instead of a full-fledged comparison function), plus a hash function, that is, a function associating integers to keys.

```
# module type HashedType =
    sig
      type t
      val equal: t → t → bool
      val hash: t → int
    end ;;
module type HashedType =
  sig type t val equal : t -> t -> bool val hash : t -> int end
# module IntMod13 =
    struct
      type t = int
      let equal = (=)
      let hash x = x mod 13
    end ;;
module IntMod13 :
  sig type t = int val equal : 'a -> 'a -> bool val hash : int -> int end
# module TblInt = Hashtbl.Make (IntMod13) ;;
module TblInt :
  sig
    type key = IntMod13.t
    and 'a t = 'a Hashtbl.Make(IntMod13).t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
  end
```

Local Module Definitions

The Objective Caml core language allows a module to be defined locally to an expression.

Syntax :

```
let module Name = structure
  in expr
```

For instance, we can use the `Set` module locally to write a sort function over integer lists, by inserting each list element into a set and finally converting the set to the sorted list of its elements.

```
# let sort l =
    let module M =
       struct
         type t = int
         let compare x y =
           if x < y then -1 else if x > y then 1 else 0
       end
   in
     let module MSet = Set.Make(M)
     in  MSet.elements (List.fold_right MSet.add l MSet.empty) ;;
val sort : int list -> int list = <fun>

# sort [ 5 ; 3 ; 8 ; 7 ; 2 ; 6 ; 1 ; 4 ] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

Objective Caml does not allow a value to escape a `let module` expression if the type of the value is not known outside the scope of the expression.

```
# let test =
    let module Foo =
      struct
        type t
        let id x = (x:t)
      end
    in Foo.id ;;
Characters 15-101:
This 'let module' expression has type Foo.t -> Foo.t
In this type, the locally bound module name Foo escapes its scope
```

Extended Example: Managing Bank Accounts

We conclude this chapter by an example illustrating the main aspects of modular programming: type abstraction, multiple views of a module, and functor-based code reuse.

The goal of this example is to provide two modules for managing a bank account. One is intended to be used by the bank, and the other by the customer. The approach is to implement a general-purpose parameterized functor providing all the needed operations, then apply it twice to the correct parameters, constraining it by the signature corresponding to its final user: the bank or the customer.

Organization of the Program

The two end modules `BManager` and `CManager` are obtained by constraining the module `Manager`. The latter is obtained by applying the functor `FManager` to the modules

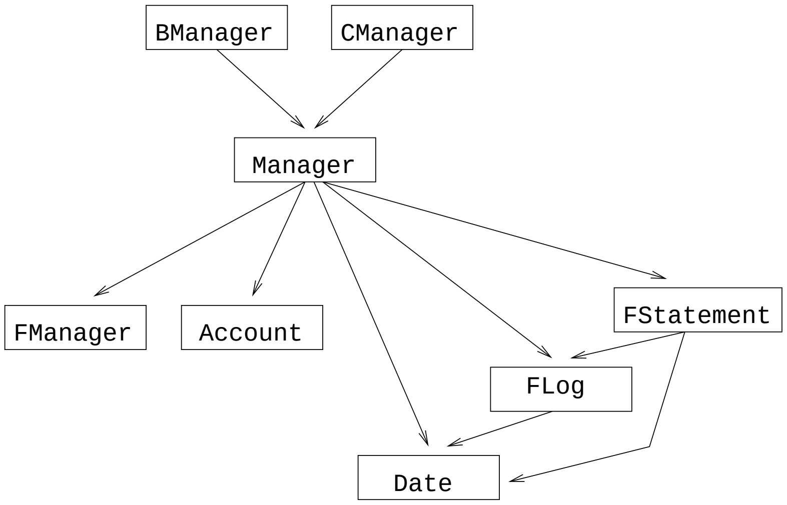

Figure 14.1: Modules dependency graph.

Account, Date and two additional modules built by application of the functors FLog and FStatement. Figure 14.1 illustrates these dependencies.

Signatures for the Module Parameters

The module for account management is parameterized by four other modules, whose signatures we now detail.

The bank account. This module provides the basic operations on the contents of the account.

```
# module type ACCOUNT = sig
    type t
    exception BadOperation
    val create : float → float → t
    val deposit : float → t → unit
    val withdraw : float → t → unit
    val balance : t → float
  end ;;
```

This set of functions provide the minimal operations on an account. The creation operation takes as arguments the initial balance and the maximal overdraft allowed. Excessive withdrawals may raise the BadOperation exception.

Ordered keys. Operations are recorded in an operation log described in the next paragraph. Each log entry is identified by a key. Key management functions are described by the following signature:

```
# module type OKEY =
    sig
      type t
      val create : unit → t
      val of_string : string → t
      val to_string : t → string
      val eq : t → t → bool
      val lt : t → t → bool
      val gt : t → t → bool
    end ;;
```

The `create` function returns a new, unique key. The functions `of_string` and `to_string` convert between keys and character strings. The three remaining functions are key comparison functions.

History. Logs of operations performed on an account are represented by the following abstract types and functions:

```
# module type LOG =
    sig
      type tkey
      type tinfo
      type t
      val create : unit → t
      val add : tkey → tinfo → t → unit
      val nth : int → t → tkey*tinfo
      val get : (tkey → bool) → t → (tkey*tinfo) list
    end ;;
```

We keep unspecified for now the types of the log keys (type *tkey*) and of the associated data (type *tinfo*), as well as the data structure for storing logs (type *t*). We assume that new informations added with the `add` function are kept in sequence. Two access functions are provided: access by position in the log (function `nth`) and access following a search predicate on keys (function `get`).

Account statements. The last parameter of the manager module provides two functions for editing a statement for an account:

```
# module type STATEMENT =
     sig
       type tdata
       type tinfo
       val editB : tdata → tinfo
       val editC : tdata → tinfo
     end ;;
```

We leave abstract the type of data to process (*tdata*) as well as the type of informations extracted from the data (*tinfo*).

The Parameterized Module for Managing Accounts

Using only the information provided by the signatures above, we now define the general-purpose functor for managing accounts.

```
# module FManager =
    functor (C:ACCOUNT) →
    functor (K:OKEY) →
    functor (L:LOG with type tkey=K.t and type tinfo=float) →
    functor (S:STATEMENT with type tdata=L.t and type tinfo
            = (L.tkey*L.tinfo) list) →
      struct
        type t = { accnt : C.t; log : L.t }
        let create s d = { accnt = C.create s d; log = L.create() }
        let deposit s g =
          C.deposit s g.accnt ; L.add (K.create()) s g.log
        let withdraw s g =
          C.withdraw s g.accnt ; L.add (K.create()) (-.s) g.log
        let balance g = C.balance g.accnt
        let statement edit g =
          let f (d,i) = (K.to_string d) ^ ":" ^ (string_of_float i)
          in List.map f (edit g.log)
        let statementB = statement S.editB
        let statementC = statement S.editC
      end ;;
module FManager :
  functor(C : ACCOUNT) ->
    functor(K : OKEY) ->
      functor
        (L : sig
               type tkey = K.t
               and tinfo = float
               and t
               val create : unit -> t
               val add : tkey -> tinfo -> t -> unit
               val nth : int -> t -> tkey * tinfo
               val get : (tkey -> bool) -> t -> (tkey * tinfo) list
             end) ->
        functor
          (S : sig
                 type tdata = L.t
                 and tinfo = (L.tkey * L.tinfo) list
                 val editB : tdata -> tinfo
                 val editC : tdata -> tinfo
               end) ->
          sig
            type t = { accnt: C.t; log: L.t }
            val create : float -> float -> t
            val deposit : L.tinfo -> t -> unit
            val withdraw : float -> t -> unit
            val balance : t -> float
            val statement : (L.t -> (K.t * float) list) -> t -> string list
            val statementB : t -> string list
            val statementC : t -> string list
          end
```

Sharing between types. The type constraint over the parameter `L` of the `FManager` functor indicates that the keys of the log are those provided by the `K` parameter, and that the informations stored in the log are floating-point numbers (the transaction amounts). The type constraint over the `S` parameter indicates that the informations contained in the statement come from the log (the `L` parameter). The signature inferred for the `FManager` functor reflects the type sharing constraints in the inferred signatures for the functor parameters.

The type `t` in the result of `FManager` is a pair of an account (`C.t`) and its transaction log.

Operations. All operations defined in this functor are defined in terms of lower-level functions provided by the module parameters. The creation, deposit and withdrawal operations affect the contents of the account and add an entry in its transaction log. The other functions return the account balance and edit statements.

Implementing the Parameters

Before building the end modules, we must first implement the parameters to the `FManager` module.

Accounts. The data structure for an account is composed of a float representing the current balance, plus the maximum overdraft allowed. The latter is used to check withdrawals.

```
# module Account:ACCOUNT =
   struct
    type t = { mutable balance:float; overdraft:float }
    exception BadOperation
    let create b o = { balance=b; overdraft=(-. o) }
    let deposit s c =  c.balance <- c.balance +. s
    let balance c = c.balance
    let withdraw s c =
     let ss = c.balance -. s in
      if ss < c.overdraft then raise BadOperation
      else c.balance <- ss
   end ;;
module Account : ACCOUNT
```

Choosing log keys. We decide that keys for transaction logs should be the date of the transaction, expressed as a floating-point number as returned by the `time` function from module `Unix`.

```
# module Date:OKEY =
   struct
    type t = float
    let create() = Unix.time()
    let of_string = float_of_string
    let to_string = string_of_float
    let eq = (=)
    let lt = (<)
    let gt = (>)
   end ;;
module Date : OKEY
```

The log. The transaction log depends on a particular choice of log keys. Hence we define logs as a functor parameterized by a key structure.

```
# module FLog (K:OKEY) =
   struct
    type tkey = K.t
    type tinfo = float
    type t = { mutable contents : (tkey*tinfo) list }
    let create() = { contents = [] }
    let add c i l = l.contents <- (c,i) :: l.contents
    let nth i l = List.nth l.contents i
    let get f l = List.filter (fun (c,_) → (f c)) l.contents
   end ;;
module FLog :
  functor(K : OKEY) ->
    sig
      type tkey = K.t
      and tinfo = float
      and t = { mutable contents: (tkey * tinfo) list }
      val create : unit -> t
      val add : tkey -> tinfo -> t -> unit
      val nth : int -> t -> tkey * tinfo
      val get : (tkey -> bool) -> t -> (tkey * tinfo) list
    end
```

Notice that the type of informations stored in log entries must be consistent with the type used in the account manager functor.

Statements. We define two functions for editing statements. The first (`editB`) lists the five most recent transactions, and is intended for the bank; the second (`editC`) lists all transactions performed during the last 10 days, and is intended for the customer.

```
# module FStatement (K:OKEY) (L:LOG with type tkey=K.t) =
   struct
    type tdata = L.t
    type tinfo = (L.tkey*L.tinfo) list
    let editB h =
     List.map (fun i → L.nth i h) [0;1;2;3;4]
    let editC h =
     let c0 = K.of_string (string_of_float ((Unix.time()) -. 864000.)) in
     let f = K.lt c0 in
      L.get f h
   end ;;
module FStatement :
  functor(K : OKEY) ->
    functor
      (L : sig
             type tkey = K.t
             and tinfo
             and t
             val create : unit -> t
             val add : tkey -> tinfo -> t -> unit
             val nth : int -> t -> tkey * tinfo
             val get : (tkey -> bool) -> t -> (tkey * tinfo) list
           end) ->
      sig
        type tdata = L.t
        and tinfo = (L.tkey * L.tinfo) list
        val editB : L.t -> (L.tkey * L.tinfo) list
        val editC : L.t -> (L.tkey * L.tinfo) list
      end
```

In order to define the 10-day statement, we need to know exactly the implementation of keys as floats. This arguably goes against the principles of type abstraction. However, the key corresponding to ten days ago is obtained from its string representation by calling the `K.of_string` function, instead of directly computing the internal representation of this date. (Our example is probably too simple to make this subtle distinction obvious.)

End modules. To build the modules `MBank` and `MCustomer`, for use by the bank and the customer respectively, we proceed as follows:

1. define a common "account manager" structure by application of the `FManager` functor;
2. declare two signatures listing only the functions accessible to the bank or to the customer;
3. constrain the structure obtained in 1 with the signatures declared in 2.

```
# module Manager =
   FManager (Account)
            (Date)
            (FLog(Date))
            (FStatement (Date) (FLog(Date))) ;;
module Manager :
  sig
    type t =
      FManager(Account)(Date)(FLog(Date))(FStatement(Date)(FLog(Date))).t =
      { accnt: Account.t;
        log: FLog(Date).t }
    val create : float -> float -> t
    val deposit : FLog(Date).tinfo -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statement :
      (FLog(Date).t -> (Date.t * float) list) -> t -> string list
    val statementB : t -> string list
    val statementC : t -> string list
  end

# module type MANAGER_BANK =
    sig
     type t
     val create : float → float → t
     val deposit : float → t → unit
     val withdraw : float → t → unit
     val balance : t → float
     val statementB : t →  string list
    end ;;

# module MBank = (Manager:MANAGER_BANK with type t=Manager.t) ;;
module MBank :
  sig
    type t = Manager.t
    val create : float -> float -> t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementB : t -> string list
  end

# module type MANAGER_CUSTOMER =
    sig
     type t
     val deposit : float → t → unit
     val withdraw : float → t → unit
     val balance : t → float
     val statementC : t →  string list
    end ;;

# module MCustomer = (Manager:MANAGER_CUSTOMER with type t=Manager.t) ;;
module MCustomer :
  sig
    type t = Manager.t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementC : t -> string list
  end
```

In order for accounts created by the bank to be usable by clients, we added the type constraint on *Manager.t* in the definition of the `MBank` and `MCustomer` structures, to ensure that their *t* type components are compatible.

Exercises

Association Lists

In this first simple exercise, we will implement a polymorphic abstract type for association lists, and present two different views of the implementation.

1. Define a signature `ALIST` declaring an abstract type with two type parameters (one for the keys, the other for the associated values), a creation function, an add function, a lookup function, a membership test, and a deletion function. The interface should be functional, *i.e.* without in-place modifications of the abstract type.
2. Define a module `Alist` implementing the signature `ALIST`
3. Define a signature `ADM_ALIST` for "administrators" of association lists. Administrators can only create association lists, and add or remove entries from a list.
4. Define a signature `USER_ALIST` for "users" of association lists. Users can only perform lookups and membership tests.
5. Define two modules `AdmAlist` and `UserAlist` for administrators and for users. Keep in mind that users must be able to access lists created by administrators.

Parameterized Vectors

This exercise illustrates the genericity and code reuse abilities of parameterized modules. We will define a functor for manipulating two-dimensional vectors (pairs of (x, y) coordinates) that can be instantiated with different types for the coordinates.

Numbers have the following signature:

```
# module type NUMBER =
   sig
    type a
    type t
    val create : a → t
    val add : t → t → t
    val string_of : t → string
```

```
  end ;;
```

1. Define the functor `FVector`, parameterized by a module of signature `NUMBER`, and defining a type *t* of two-dimensional vectors over these numbers, a creation function, an addition function, and a conversion to strings.
2. Define a signature `VECTOR`, without parameters, where the types of numbers and vectors are abstract.
3. Define three structures `Rational`, `Float` et `Complex` implementing the signature `NUMBER`.
4. Use these structures to define (by functor application) three modules for vectors of rationals, reals and complex.

Lexical Trees

This exercise follows up on the lexical trees introduced in chapter 2, page 63. The goal is to define a generic module for handling lexical trees, parameterized by an abstract type of words.

1. Define the signature `WORD` defining an abstract type *alpha* for letters of the alphabet, and another abstract type *t* for words on this alphabet. Declare also the empty word, the conversion from an alphabet letter to a one-letter word, the accessor to a letter of a word, the sub-word operation, the length of a word, and word concatenation.
2. Define the functor `LexTree`, parameterized by a module implementing `WORD`, that defines (as a function of the types and operations over words) the type of lexical trees and functions `exists`, `insert` et `select` similar to those from chapter 2, page 63.
3. Define the module `Chars` implementing the `WORD` signature for the types *alpha = char* and *t = string*. Use it to obtain a module `CharDict` implementing dictionaries whose keys are character strings.

Summary

In this chapter, we introduced all the facilities that the Objective Caml module language offers, in particular parameterized modules.

As all module systems, it reflects the duality between interfaces and implementations, here presented as a duality between signatures and structures. Signatures allow hiding information about type, value or exception definitions.

By hiding type representation, we can make certain types abstract, ensuring that values of these types can only be manipulated through the operations provided in the module signature. We saw how to exploit this mechanism to facilitate sharing of values hidden in closures, and to offer multiple views of a given implementation. In the latter case, explicit type sharing annotations are sometimes necessary to achieve the desired behavior.

Parameterized modules, also called functors, go one step beyond and support code reuse through simple mechanisms similar to function abstraction and function application.

To Learn More

Other examples of modules and functors can be found in chapter 4 of the Objective Caml manual.

The underlying theory and the type checking for modules can be found in a number of research articles and course notes by Xavier Leroy, at

Link: http://cristal.inria.fr/~xleroy

The Objective Caml module system follows the same principles as that of its cousin the SML language. Chapter 22 compares these two languages in more details and provides bibliographical references for the interested reader.

Other languages feature advanced module systems, in particular Modula-3 (2 and 3), and ADA. They support the definition of modules parameterized by types and values.