7
Abstract Types and Functors

Everyone accepts that large programs should be organized as hierarchical modules. Standard ML's structures and signatures meet this requirement. Structures let us package up declarations of related types, values and functions. Signatures let us specify what components a structure must contain. Using structures and signatures in their simplest form we have treated examples ranging from the complex numbers in Chapter 2 to infinite sequences in Chapter 5.

A modular structure makes a program easier to understand. Better still, the modules ought to serve as interchangeable parts: replacing one module by an improved version should not require changing the rest of the program. Standard ML's ***abstract types*** and ***functors*** can help us meet this objective too.

A module may reveal its internal details. When the module is replaced, other parts of the program that depend upon such details will fail. ML provides several ways of declaring an abstract type and related operations, while hiding the type's representation.

If structure B depends upon structure A, and we wish to replace A by another structure A', we could edit the program text and recompile the program. That is satisfactory if A is obsolete and can be discarded. But what if A and A' are both useful, such as structures for floating point arithmetic in different precisions?

ML lets us declare B to take a structure as a parameter. We can then invoke $B(A)$ and $B(A')$, possibly at the same time. A parametric structure, such as B, is called a ***functor***. Functors let us treat A and A' as interchangeable parts.

The language of modules is distinct from the core language of types and expressions. It is concerned with program organization, not with computation itself. Modules may contain types and expressions, but not the other way around. The main module constructs have counterparts in the core language:

$$\begin{aligned}\text{structure} &\sim \text{value}\\ \text{signature} &\sim \text{type}\\ \text{functor} &\sim \text{function}\end{aligned}$$

This analogy is a starting point to understanding, but it fails to convey the full potential of ML modules.

Chapter outline

This chapter examines structures and signatures in greater depth, and introduces abstract types and functors. Many features of the module language are provided mainly to support functors. The chapter contains the following sections:

Three representations of queues. Three different structures implement queues, illustrating the idea of multiple data representations. But structures do not hide the representation of queues; it could be abused elsewhere in the program.

Signatures and abstraction. Signature constraints on the queue structures can hide details, declaring an abstract type of queues. The `abstype` declaration is a more flexible means of declaring abstract types. The three queue representations have their own concrete signatures.

Functors. Functors let us use the three queue implementations as interchangeable parts, first in a test harness and then for breadth-first search. Another example is generic matrix arithmetic, with numerical and graph applications. Functors allow dictionaries and priority queues to be expressed generically, for an arbitrary ordered type.

Building large systems using modules. A variety of deeper topics are covered: multiple arguments to functors, sharing constraints and the fully-functorial programming style. New declaration forms, such as `open` and `include`, help manage the deep hierarchies found in large programs.

Reference guide to modules. The full modules language is presented systematically and concisely.

Three representations of queues

A ***queue*** is a sequence whose elements may be inserted only at the end and removed only from the front. Queues enforce a first-in-first-out (FIFO) discipline. They provide the following operations:

- $empty$: the empty queue
- $enq(q, x)$: the queue obtained by inserting x on the end of q
- $null(q)$: the boolean-valued test of whether q is empty
- $hd(q)$: the front element of q
- $deq(q)$: the queue obtained by removing the front element of q
- E: the exception raised by hd and deq if the queue is empty

The queue operations are functional; *enq* and *deq* create new queues rather than modifying existing queues. We shall discuss several ways of representing queues and defining their operations as ML structures, eventually finding an efficient representation.

The names *enq* and *deq* abbreviate the words enqueue and dequeue, while *null* and *hd* clash with existing list operations. As we shall package the operations into structures, we can get away with short names without concern for clashes.

It is a simple exercise to write down the corresponding signature, but let us defer signatures until the next section. Then we shall also consider how to hide the representation of queues by declaring abstract data types.

7.1 *Representing queues as lists*

Representation 1, perhaps the most obvious, maintains a queue as the list of its elements. The structure *Queue*1 is declared as follows:

```
structure Queue1 =
  struct
  type 'a t = 'a list;
  exception E;

  val empty = [];

  fun enq(q,x) = q @ [x];

  fun null(x::q) = false
    | null _      = true;

  fun hd(x::q) = x
    | hd []     = raise E;

  fun deq(x::q) = q
    | deq []     = raise E;
  end;
```

The type of queues is simply $\alpha\ t$; outside the structure it is $\alpha\ Queue1.t$. The type abbreviation makes $\alpha\ Queue1.t$ a synonym for $\alpha\ list$. (Recall how in Section 2.7 we made *vec* a synonym for $real \times real$.) Since a value of type $\alpha\ Queue1.t$ can be used with any list operations, the type name is little more than a comment.

Function *enq* uses append, while *deq* uses pattern-matching. The other queue operations are implemented easily and efficiently. But $enq(q, x)$ takes time proportional to the length of q: quite unsatisfactory.

Structures do not hide information. Declaring a structure hardly differs from declaring its items separately, except that a structure declaration is taken as a unit and introduces compound names. Each item behaves as if it were declared separately. Structure *Queue*1 makes no distinction between queues and lists:

```
Queue1.deq ["We","happy","few"];
> ["happy", "few"] : string list
```

7.2 *Representing queues as a new datatype*

Representation 2 declares a datatype with constructors *empty* and *enq*. The operation $enq(q, x)$ now takes constant time, independent of the length of q, but $hd(q)$ and $deq(q)$ are slow. Calling $deq(q)$ copies the remaining elements of q. Even $hd(q)$ requires recursive calls.

```
structure Queue2 =
  struct
  datatype 'a t = empty
                | enq of 'a t * 'a;
  exception E;

  fun null (enq _) = false
    | null empty   = true;

  fun hd  (enq(empty,x)) = x
    | hd  (enq(q,x))     = hd q
    | hd  empty          = raise E;

  fun deq  (enq(empty,x)) = empty
    | deq  (enq(q,x))     = enq(deq q, x)
    | deq  empty          = raise E;
  end;
```

Representation 2 gains little by defining a new datatype. It is essentially no different from representing a queue by a reversed list. Then

$$enq(q, x) = x :: q,$$

while deq is a recursive function to remove the last element from a list. We could call this Representation 2a.

The type of queues, α *Queue*2.*t*, is not abstract: it is a datatype with constructors *Queue*2.*empty* and *Queue*2.*enq*. Pattern-matching with the constructors can remove the last element of the queue, violating its FIFO discipline:

```
fun last (Queue2.enq(q,x)) = x;
> val last = fn : 'a Queue2.t -> 'a
```

Such declarations abuse the data structure. Enough of them scattered throughout a program can make it virtually impossible to change the representation of queues. The program can no longer be maintained.

Again, structures do not hide information. The differences between $Queue1$ and $Queue2$ are visible outside. The function $Queue1.null$ may be applied to any list, while $Queue2.null$ may only be applied to values of type $\alpha\ Queue2.t$. Both $Queue1.enq$ and $Queue2.enq$ are functions, but $Queue2.enq$ is a constructor and may appear in patterns.

Our `datatype` declaration flouts the convention that constructor names start with a capital letter (Section 4.4). Within the confines of a small structure this is a minor matter, but to export such a constructor is questionable.

7.3 *Representing queues as pairs of lists*

Representation 3 (Burton, 1982) maintains a queue as a pair of lists. The pair

$$([x_1, x_2, \ldots, x_m],\ [y_1, y_2, \ldots, y_n])$$

denotes the queue

$$x_1 x_2 \cdots x_m y_n \cdots y_2 y_1.$$

The queue has a front part and a rear part. The elements of the rear part are stored in reverse order so that new ones can quickly be added to the end of the queue; $enq(q, y)$ modifies the queue thus:

$$(xs,\ [y_1, \ldots, y_n]) \mapsto (xs,\ [y, y_1, \ldots, y_n])$$

The elements of the front part are stored in correct order so that they can quickly be removed from the queue; $deq(q)$ modifies the queue thus:

$$([x_1, x_2, \ldots, x_m],\ ys) \mapsto ([x_2, \ldots, x_m],\ ys)$$

When the front part becomes empty, the rear part is reversed and moved to the front:

$$([],\ [y_1, y_2, \ldots, y_n]) \mapsto ([y_n, \ldots, y_2, y_1],\ [])$$

The rear part then accumulates further elements until the front part is again emptied. A queue is in ***normal form*** provided it does not have the form

$$([],\ [y_1, y_2, \ldots, y_n])$$

for $n \geq 1$. The queue operations ensure that their result is in normal form. Therefore, inspecting the first element of a queue does not perform a reversal. A normal queue is empty if its front part is empty.

Here is a structure for this approach. The type of queues is declared as a datatype with one constructor, not as a type abbreviation. We used a similar technique for flexible arrays (page 158). The constructor costs nothing at run-time, while making occurrences of queues stand out in the code.

```
structure Queue3 =
  struct
  datatype 'a t = Queue of ('a list * 'a list);
  exception E;

  val empty = Queue([],[]);

  fun norm (Queue([],tails)) = Queue(rev tails, [])
    | norm q                 = q;

  fun enq(Queue(heads,tails), x) = norm(Queue(heads, x::tails));

  fun null(Queue([],[])) = true
    | null _             = false;

  fun hd(Queue(x::_,_)) = x
    | hd(Queue([],_))   = raise E;

  fun deq(Queue(x::heads,tails)) = norm(Queue(heads,tails))
    | deq(Queue([],_))           = raise E;
  end;
```

The function *norm* puts a queue into normal form by reversing the rear part, if necessary. It is called by *enq* and *deq*, since a queue must be put into normal form every time an element is added or removed.

Once again, none of the internal details are hidden. Users can tamper with the constructor *Queue3.Queue* and the function *Queue3.norm*. Pattern-matching with the constructor *Queue* exposes a queue as consisting of a pair of lists. Inside the structure, such access is essential; used outside, it could violate the queue's FIFO discipline. Calling *Queue3.norm* from outside the structure can serve no purpose.

How efficient is this representation? The use of reverse may seem expensive. But the cost of an *enq* or *deq* operation is constant when averaged over the lifetime of the queue. At most two cons (::) operations are performed per queue element, one when it is put on the rear part and one when it is moved to the front part.

Measuring a cost over the lifetime of the data structure is called an ***amortized*** cost (Cormen *et al.*, 1990). Sleator and Tarjan (1985) present another data structure, self-adjusting trees, designed for a good amortized cost. The main drawback of such a data structure is that the costs are not evenly distributed. When normalization takes place, the reverse operation could cause an unexpected delay.

Also, the amortized cost calculation assumes that the queue usage follows an imperative style: is ***single-threaded***. Every time the data structure is updated, the previous value should be discarded. If we violate this assumption by repeatedly applying *deq* to a queue of the form

$$([x],\ [y_1, y_2, \ldots, y_n])$$

then additional normalizations will result, incurring greater costs.

Flexible arrays can represent queues without these drawbacks. But the cost of each operation is greater, order log n where n is the number of elements in the queue. Representation 3 is simple and efficient, and can be recommended for most situations requiring functional queues.

Exercise 7.1 Under Representation 1, how much time does it take to build an n-element queue by applying enq operations to the empty queue?

Exercise 7.2 Discuss the relative merits of the three representations of functional queues. For example, are there any circumstances under which Representation 1 might be more efficient than Representation 3?

Exercise 7.3 Code Representation 2a in ML.

Exercise 7.4 Representation 4 uses flexible arrays, with $hiext$ implementing enq and $lorem$ implementing deq. Code Representation 4 and compare its efficiency with Representation 3.

Exercise 7.5 A queue is conventionally represented using an array, with indices to its first and last elements. Are the functional arrays of Chapter 4 suitable for this purpose? How would it compare with the other representations of functional queues?

Signatures and abstraction

An ***abstract type*** is a type equipped with a set of operations, which are the only operations applicable to that type. Its representation can be changed — perhaps to a more efficient one — without affecting the rest of the program. Abstract types make programs easier to understand and modify. Queues should be defined as an abstract type, hiding internal details.

We can limit outside access to the components of a structure by constraining its signature. We can hide a type's representation by means of an `abstype` declaration. Combining these methods yields abstract structures.

7.4 *The intended signature for queues*

Although the structures $Queue1$, $Queue2$ and $Queue3$ differ, they each implement queues. Moreover they share a common interface, defined by signature *QUEUE*:

```
signature QUEUE =
  sig
  type 'a t                        (*type of queues*)
  exception E                      (*for errors in hd, deq*)
  val empty: 'a t                  (*the empty queue*)
  val enq  : 'a t * 'a -> 'a t     (*add to end*)
  val null : 'a t -> bool          (*test for empty queue*)
  val hd   : 'a t -> 'a            (*return front element*)
  val deq  : 'a t -> 'a t          (*remove from front*)
  end;
```

Each entry in a signature is called a ***specification***. The comments after each specification are optional, but make the signature more informative. A structure is an ***instance*** of this signature provided it declares, at least,

- a polymorphic type $\alpha\ t$ (which need not admit equality)
- an exception E
- a value $empty$ of type $\alpha\ t$
- a value enq of type $\alpha\ t \times \alpha \to \alpha\ t$
- a value $null$ of type $\alpha\ t \to bool$
- a value hd of type $\alpha\ t \to \alpha$
- a value deq of type $\alpha\ t \to \alpha\ t$

Consider each structure in turn. In $Queue1$, type $\alpha\ t$ abbreviates $\alpha\ list$, and the values have the correct types under this abbreviation. In $Queue2$, type $\alpha\ t$ is a datatype and $empty$ and enq are constructors. In $Queue3$, type $\alpha\ t$ is again a datatype; the structure declares everything required by signature *QUEUE*, and the additional items $Queue$ and $norm$. An instance of a signature may contain items not specified in the signature.

7.5 *Signature constraints*

Different views of a structure, with varying degrees of abstraction, can be obtained using different signatures. A structure can be constrained to a signature either when it is first defined or later. A constraint can be transparent or opaque.

Transparent signature constraints. The constraints we have used until now, indicated by a colon (:), are transparent. To see what this implies, let us constrain our existing queue structures using signature *QUEUE*:

```
structure S1: QUEUE = Queue1;
structure S2: QUEUE = Queue2;
structure S3: QUEUE = Queue3;
```

These declarations make $S1$, $S2$ and $S3$ denote the same structures as $Queue1$, $Queue2$ and $Queue3$, respectively. However, the new structures are constrained to have the signature *QUEUE*. The types α $Queue2.t$ and α $S2.t$ are identical, yet $Queue2.empty$ is a constructor while $S2.empty$ may only be used as a value. The structures $Queue3$ and $S3$ are identical, yet $Queue3.norm$ is a function while $S3.norm$ means nothing.

A transparent signature constraint may hide components, but they are still present. This cannot be called abstraction. Structure $S1$ does not hide its representation at all; type α $S1.t$ is identical to α *list*.

```
S1.deq ["We","band","of","brothers"];
> ["band", "of", "brothers"] : string S1.t
```

Structures $S2$ and $S3$ may seem more abstract, because they declare the type α t and hide its constructors. Without the constructors, pattern-matching is not available to take apart values of the type and disclose the representation. However, the constructor $Queue3.Queue$ may be used in a pattern to take apart a value of type α $S3.t$:

```
val Queue3.Queue(heads,tails) =
    S3.enq(S3.enq(S3.empty,"Saint"), "Crispin");
> val heads = ["Saint"] : string list
> val tails = ["Crispin"] : string list
```

The concrete structure, $Queue3$, provides a loophole into its abstract view, $S3$.

Data abstraction is compromised in another way. For each of our queue structures, type α t admits equality testing. The equality test compares internal representations, not queues. Under Representation 3, the values ([1, 2], []) and ([1], [2]) denote the same queue, but the equality test says they are different.

Opaque signature constraints. Using the symbol :> instead of a colon makes the constraint opaque. The constraint hides all information about the new structure except its signature. Let us create some truly abstract queue structures by constraining the concrete ones:

```
structure AbsQueue1 :> QUEUE = Queue1;
structure AbsQueue2 :> QUEUE = Queue2;
structure AbsQueue3 :> QUEUE = Queue3;
```

The components of the constrained structure are divorced from their counterparts in the original structure. Structure $AbsQueue1$ represents queues by lists, but we cannot see this:

```
AbsQueue1.deq ["We","band","of","brothers"];
> Error: Type conflict:...
```

Type checking similarly forbids using the constructor *Queue*3.*Queue* to take apart the queues of structure *AbsQueue*3. Equality testing is forbidden too:

```
AbsQueue3.empty = AbsQueue3.empty;
> Error: type 'a AbsQueue3.t must be an equality type
```

Specifying a type by `eqtype` *t* instead of `type` *t* indicates that the type is to admit equality. Using `eqtype` in the signature allows you to export the type's equality test, even with an opaque signature constraint.

Limitations. An opaque signature constraint is perfect for declaring an abstract type of queues. The abstract structure can be made from an existing concrete structure, as in the *AbsQueue* declarations above, or we can simply constrain the original structure declaration:

```
structure Queue :> QUEUE = struct ... end;
```

But the two kinds of signature constraints give us an all-or-nothing choice, which is awkward for complex abstract types. Signature *DICTIONARY* specifies two types: *key* is the type of search keys; α *t* is the type of dictionaries (see Section 4.14). Type α *t* should be abstract, but *key* should be something concrete, like *string*. Otherwise, we should have no way to refer to keys; we should be unable to call *lookup* and *update*! The next section describes a more flexible approach to declaring abstract types.

Exercise 7.6 Assuming Representation 3, show how two different representations of the same queue value could be created using only the abstract queue operations.

Exercise 7.7 Extend signature *QUEUE* to specify the functions *length*, for returning the number of elements in a queue, and *equal*, for testing whether two queues consist of the same sequence of elements. Extend the structures *Queue*1, *Queue*2 and *Queue*3 with declarations of these functions.

7.6 *The `abstype` declaration*

Standard ML has a declaration form specifically intended for declaring abstract types. It hides the representation fully, including the equality test. The `abstype` declaration originates from the first ML dialect and reflects the early thinking of the structured programming school. Now it looks distinctly dated.

But it is more selective than an opaque constraint: it applies to chosen types instead of an entire signature.

A simple `abstype` declaration contains two elements, a datatype binding DB and a declaration D:

```
abstype DB with D end
```

A datatype binding is a type name followed by constructor descriptions, exactly as they would appear in a `datatype` declaration. The constructors are visible within the declaration part, D, which must use them to implement all operations associated with the abstract type. Identifiers declared in D are visible outside, as is the type, but its constructors are hidden. Moreover, the type does not admit equality testing.

To illustrate the `abstype` declaration, let us apply it to queues. The declaration ought to be enclosed in a structure to prevent name clashes with the built-in list functions $null$ and hd. But as a structure would complicate the example, those functions have instead been renamed. Exceptions are omitted to save space.

Queues as lists. We begin with Representation 1. Although $list$ is already a datatype, the `abstype` declaration forces us to use a new constructor ($Q1$) in all the queue operations. This constructor is traditionally called the ***abstraction function***, as it maps concrete representations to abstract values.

```
abstype 'a queue1 = Q1 of 'a list
  with
  val empty = Q1 [];

  fun enq(Q1 q, x) = Q1 (q @ [x]);

  fun qnull(Q1(x::q)) = false
    | qnull _         = true;

  fun qhd(Q1(x::q)) = x;

  fun deq(Q1(x::q)) = Q1 q;
  end;
```

In its response, ML echoes the names and types of the identifiers that have been declared:

```
> type 'a queue1
> val empty = - : 'a queue1
> val enq = fn : 'a queue1 * 'a -> 'a queue1
> val qnull = fn : 'a queue1 -> bool
> val qhd = fn : 'a queue1 -> 'a
> val deq = fn : 'a queue1 -> 'a queue1
```

The abstype declaration has hidden the connection between *queue*1 and *list*.

Queues as a new datatype. Now turn to Representation 2. Previously we called the constructors *empty* and *enq*, with lower case names, for use outside as values. And that was naughty. But the abstype declaration hides the constructors. We may as well give them capitalised names *Empty* and *Enq*, since we must now export their values explicitly:

```
abstype 'a queue2 = Empty
                  | Enq of 'a queue2 * 'a
  with
  val empty = Empty
  and enq   = Enq

  fun qnull (Enq _) = false
    | qnull Empty   = true;

  fun qhd (Enq(Empty,x)) = x
    | qhd (Enq(q,x))     = qhd q;

  fun deq (Enq(Empty,x)) = Empty
    | deq (Enq(q,x))     = Enq(deq q, x);
  end;
```

We do not need to declare a new constructor *Q*2 because this representation requires its own constructors. ML's response is identical to its response to the declaration of *queue*1 except for the name of the queue type. An external user can operate on queues only by the exported operations.

These two examples illustrate the main features of abstype. We do not need to see the analogous declaration of *queue*3.

Abstract types in ML: summary. ML's treatment of abstract types is less straightforward than one might like, but it can be reduced to a few steps. If you would like to declare a type t and allow access only by operations you have chosen to export, here is how to proceed.

1 Consider whether to export the equality test for t. It is only appropriate if the representation admits equality, and if this equality coincides with equality of the abstract values. Also consider whether equality testing would actually be useful. Equality testing is appropriate for small objects such as dates and rational numbers, but not for matrices or flexible arrays.

2 Declare a signature *SIG* specifying the abstract type and its operations. The signature must specify *t* as an `eqtype` if it is to admit equality, and as a `type` otherwise.
3 Decide which sort of signature constraint to use with *SIG*. An opaque constraint is suitable only if all the types in the signatures are intended to be abstract.
4 Write the shell of a structure (or functor) declaration, attaching the constraint chosen in the previous step.
5 Within the brackets `struct` and `end`, declare type *t* and the desired operations. If you used a transparent signature constraint, this must be either a `datatype` declaration (to export equality) or an `abstype` declaration (to hide equality).

A `datatype` declaration can yield an abstract type because the signature constraint hides the constructors. An `abstype` or `datatype` declaration creates a fresh type, which ML regards as distinct from all others.

Functor *Dictionary* exemplifies the first approach (see page 286). The ring buffer structure *RingBuf* exemplifies the second (see page 339).

Exercise 7.8 Early papers on abstract types all considered the same example: stacks. The operations included *push* (which puts an item on top of the stack), *top* (which returns the top item) and *pop* (which discards the top item). At least two other operations are needed. Complete the design and code two distinct representations using `abstype`.

Exercise 7.9 Write an `abstype` declaration for the rational numbers, following Exercise 2.25 on page 63. Use a `local` declaration to keep any auxiliary functions private. Then modify your solution to obtain a structure matching signature *ARITH*.

Exercise 7.10 Design and code an `abstype` declaration for type *date*, which represents dates as a day and a month. (Assume it is not a leap year.) Provide a function *today* for converting a valid day and month to a date. Provide functions *tomorrow* and *yesterday*; they should raise an exception if the desired date lies outside the current year.

7.7 *Inferred signatures for structures*

A structure declaration can appear without a signature constraint, as in the declarations of *Queue*1, *Queue*2 and *Queue*3. ML then infers a signature fully describing the structure's internal details.

Signature *QUEUE1* is equivalent to the signature that is inferred for structure *Queue*1. It specifies $\alpha\ t$ as an `eqtype` — a type that admits equality — because lists can be compared for equality. Observe that the types of values involve type $\alpha\ list$ instead of $\alpha\ t$, as in signature *QUEUE*.

```
signature QUEUE1 =
  sig
  eqtype 'a t
  exception E
  val empty : 'a list
  val enq   : 'a list * 'a -> 'a list
  val null  : 'a list -> bool
  val hd    : 'a list -> 'a
  val deq   : 'a list -> 'a list
  end;
```

The signature inferred for *Queue*2 specifies $\alpha\ t$ as a `datatype` with constructors *empty* and *enq*; constructors are not specified again as values. The signature could be declared as follows:

```
signature QUEUE2 =
  sig
  datatype 'a t = empty | enq of 'a t * 'a
  exception E
  val null : 'a t -> bool
  val hd   : 'a t -> 'a
  val deq  : 'a t -> 'a t
  end;
```

The signature inferred for structure *Queue*3 again specifies $\alpha\ t$ as a `datatype` — not merely a type, as in signature *QUEUE*. All items in the structure are specified, including *Queue* and *norm*.

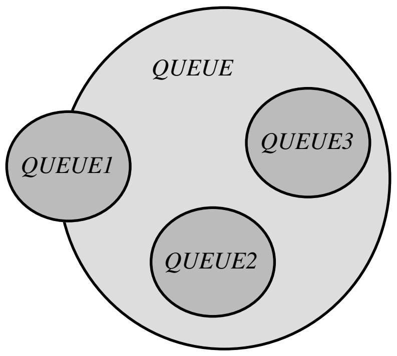
```
signature QUEUE3 =
  sig
  datatype 'a t = Queue of 'a list * 'a list
  exception E
  val empty : 'a t
  val enq   : 'a t * 'a -> 'a t
  val null  : 'a t -> bool
  val hd    : 'a t -> 'a
  val deq   : 'a t -> 'a t
  val norm  : 'a t -> 'a t
  end;
```

These signatures are more concrete and specific than *QUEUE*. No structure can be an instance of more than one of them. Consider *QUEUE1* and *QUEUE3*. Function *hd* must have type $\alpha\ list \to \alpha$ to satisfy *QUEUE1*; it must have type

$\alpha\ t \to \alpha$ to satisfy *QUEUE3*, which also specifies that $\alpha\ t$ is a datatype clearly different from $\alpha\ list$.

On the other hand, each signature has many different instances. A structure can satisfy the specification `val` $x : int$ by declaring x to be any value of type int. It can satisfy the specification `type` t by declaring t to be any type. (However, it can satisfy a `datatype` specification only by an identical `datatype` declaration.) A structure may include items not specified in the signature. Thus, a signature defines a class of structures.

Interesting relationships hold among these classes. We have already seen that *QUEUE1*, *QUEUE2* and *QUEUE3* are disjoint. The latter two are contained in *QUEUE*; an instance of *QUEUE2* or *QUEUE3* is an instance of *QUEUE*. An instance of *QUEUE1* is an instance of *QUEUE* only if it makes type $\alpha\ t$ equivalent to $\alpha\ list$. These containments can be shown in a Venn diagram:



Exercise 7.11 Declare a structure that has signature *QUEUE1* and that implements queues by a different representation from that of $Queue1$.

Exercise 7.12 Declare a structure that has signature *QUEUE* but does not implement queues. After all, the signature specifies only the types of the queue operations, not their other properties.

Functors

An ML function is an expression that takes parameters. Applying it substitutes argument values for the parameters. The value of the resulting expression is returned. A function can only be applied to arguments of the correct type.

We have several implementations of queues. Could we write code that uses queues but is independent of any particular implementation? This seems to require taking a structure as a parameter.

Functions themselves can be parameters, for functions are values in ML. Records are also values. They are a bit like structures, but they cannot represent queue implementations because they cannot have types and exception constructors as components.

An ML ***functor*** is a structure that takes other structures as parameters. Applying it substitutes argument structures for the parameters. The bindings that arise from the resulting structure are returned. A functor can only be applied to arguments that match the correct signature.

Functors let us write program units that can be combined in different ways. A replacement unit can quickly be linked in, and the new system tested. Functors can also express generic algorithms. Let us see how they do so.

7.8 *Testing the queue structures*

Here is a simple test harness for queues. Given a queue structure, it returns a testing structure containing two functions. One converts a list to a queue; the other is the inverse operation. The test harness is declared as a functor with argument signature *QUEUE*:

```
functor TestQueue (Q: QUEUE) =
  struct
  fun fromList l = foldl (fn (x,q) => Q.enq(q,x)) Q.empty l;

  fun toList q   = if Q.null q then []
                   else Q.hd q :: toList (Q.deq q);
  end;
> functor TestQueue : <sig>
```

The functor body refers not to existing queue structures but to the argument Q. The two functions exercise the queue operations uniformly. Any queue structure can be tested and its efficiency measured. Let us start with $Queue3$. Applying the functor to this argument yields a new structure, which we name $TestQ3$. The components of $TestQ3$ are functions to test $Queue3$, as can be seen from their types:

```
structure TestQ3 = TestQueue (Queue3);
> structure TestQ3 :
>   sig
>   val fromList : 'a list -> 'a Queue3.t
>   val toList   : 'a Queue3.t -> 'a list
>   end
```

The test data is just the list of integers from 1 to 10,000:

```
val ns = upto(1,10000);
> val ns = [1, 2, 3, 4, ...] : int list
val q3 = TestQ3.fromList ns;
> val q3 = Queue ([1], [10000, 9999, 9998, 9997, ...])
> : int Queue3.t
val l3 = TestQ3.toList q3;
> val l3 = [1, 2, 3, 4, ...] : int list
l3 = ns;
> true : bool
```

*Queue*3 passes its first test: we get back the original list. It is also efficient, taking 10 msec to build *q*3 and 50 msec to convert it back to a list.

ML's response to the declaration of *q*3 reveals its representation as a pair of lists: *Queue*3 does not define an abstract type. We ought to try structure *AbsQueue3*. Again we apply the functor and give the resulting structure a name:

```
structure TestAQ3 = TestQueue (AbsQueue3);
> structure TestAQ3 :
>   sig
>   val fromList : 'a list -> 'a AbsQueue3.t
>   val toList   : 'a AbsQueue3.t -> 'a list
>   end
val q = TestAQ3.fromList ns;
> val q = - : int AbsQueue3.t
```

Now ML reveals nothing about the representation. In terms of efficiency, *Queue*3 and *AbsQueue*3 are indistinguishable. Similar measurements reveal that *AbsQueue3* is orders of magnitude faster than *Queue*1 and *Queue*2 and much faster than the balanced tree representation suggested in Exercise 7.4. Because *Queue*1 represents queues by lists, it could implement *fromList* and *toList* efficiently, but only operations specified in signature *QUEUE* are allowed in the functor body.

A more realistic test would involve an application of queues, such as breadth-first search. Function *breadthFirst* (Section 5.17) used lists instead of queues, for simplicity. A functor can express the search strategy independently from the implementation of queues.

```
functor BreadthFirst (Q: QUEUE) =
  struct
  fun enqlist q xs = foldl (fn (x,q) => Q.enq(q,x)) q xs;
  fun search next x =
    let fun bfs q =
          if Q.null q then Nil else
            let val y = Q.hd q
            in  Cons(y, fn()=> bfs (enqlist (Q.deq q) (next y)))
            end
```

```
     in  bfs (Q.enq(Q.empty, x))  end;
   end;
> functor BreadthFirst : <sig>
```

The function *enqlist* appends a list of elements to a queue. Let us apply the functor to an efficient queue structure:

```
structure Breadth = BreadthFirst (Queue3);
> structure Breadth :
>   sig
>   val enqlist : 'a Queue3.t -> 'a list -> 'a Queue3.t
>   val search  : ('a -> 'a list) -> 'a -> 'a seq
>   end
```

The function *Breadth.search* is equivalent to *breadthFirst*, but runs a lot faster.

Most languages have nothing comparable to functors. The C programmer obtains a similar effect using header and include files. Primitive methods such as these go a long way, but they do not forgive errors. Including the wrong file means the wrong code is compiled: we get a cascade of error messages. What happens if a functor is applied to the wrong sort of structure? Try applying *BreadthFirst* to the standard library structure *List*:

```
structure Wrong = BreadthFirst (List);
> Error: unmatched type spec: t
> Error: unmatched exception spec: E
> Error: unmatched val spec: empty
> Error: unmatched val spec: enq
> Error: unmatched val spec: deq
```

We get specific error messages describing what is missing from the argument. There is no complaint about the absence of *hd* and *null* because *List* has components with those names.

Taking the queue structure as a parameter may be a needless complication. *AbsQueue*3 is the best queue structure; we may as well rename it *Queue* and use it directly, just as we use standard library structures such as *List*. But often we have a choice. There are several possible representations for dictionaries and priority queues. Even the standard library admits competing structures for real arithmetic. And when we consider generic operations, the case for functors becomes unassailable.

Exercise 7.13 Consider how you would obtain the effect of ML modules in another language of your choice. How would you express signatures such as *QUEUE*, alternative structures such as *Queue*1 and *Queue*2, and functors such as *TestQueue*?

Exercise 7.14 To what extent is *TestQueue* a good test suite for queues?

7.9 *Generic matrix arithmetic*

Related structures can differ in other ways than performance. In Section 2.22 we considered the signature *ARITH*, which specifies the components *zero*, *sum*, *diff*, *prod*, etc. Suitable instances of this signature include structures that implement arithmetic on integers, reals, complex numbers and rational numbers. Chapter 3 mentioned further possibilities: binary numerals, matrices and polynomials.

To illustrate functors, let us code a generic structure for matrix arithmetic. For simplicity we shall treat only zero, sum and product:

```
signature ZSP =
   sig
   type t
   val zero : t
   val sum  : t * t -> t
   val prod : t * t -> t
   end;
```

We shall declare a functor whose argument and result structures both match signature *ZSP*.

Declaring the matrix functor. Given a type t and the three arithmetic operations, functor *MatrixZSP* declares a type for matrices over t and the analogous matrix operations (Figure 7.1 on the following page). Before you study the functor body, reviewing Section 3.10 may be helpful.

In the functor heading, the second occurrence of **:** *ZSP* is a signature constraint on the result structure. Because the constraint is transparent, *MatrixZSP* does not return an abstract type. If it were opaque (using **:>**) then we could only operate on matrices using the exported operations of zero, sum and product: we could only express zero! As things stand, we can write matrices as lists of lists.

The result structure declares t, the type of matrices, in terms of the type of its elements, $Z.t$. The declaration is required by the result signature, which specifies a type t. The functor body never refers to it.

The structure then declares *zero*. In algebra, any $m \times n$ matrix of zeros is called a ***zero matrix***. The specification *zero* **:** t in signature *ZSP* requires us to declare a single zero element. So the functor declares *zero* to be the empty list, and makes *sum* and *prod* satisfy the laws $\mathbf{0} + A = A + \mathbf{0} = A$ and $\mathbf{0} \times A = A \times \mathbf{0} = \mathbf{0}$.

The structure declares the function *sum* to compute the sum of two matri-

Figure 7.1 *A functor for generic matrix arithmetic*

```
functor MatrixZSP (Z: ZSP) : ZSP =
  struct
  type t    = Z.t list list;

  val zero  = [];

  fun sum (rowsA,[])       = rowsA
    | sum ([],rowsB)       = rowsB
    | sum (rowsA,rowsB) = ListPair.map (ListPair.map Z.sum)
                                       (rowsA,rowsB);

  fun dotprod pairs = foldl Z.sum Z.zero (ListPair.map Z.prod pairs);

  fun transp ([]::_) = []
    | transp rows      = map hd rows :: transp (map tl rows);

  fun prod (rowsA,[])      = []
    | prod (rowsA,rowsB) =
        let val colsB = transp rowsB
        in  map (fn row => map (fn col => dotprod(row,col))
                               colsB)
                rowsA
        end;
  end;
```

ces. Two rows are added by adding corresponding elements, using the library functional *ListPair*.*map*. Two matrices are added similarly, by adding corresponding rows. There is no conflict between *sum* (matrix addition) and *Z*.*sum* (element addition).

Other functions in the structure support the declaration of *prod*. The dot product computation is also streamlined by *ListPair*.*map*, while matrix transpose is declared as in Section 5.7. As *transp* cannot handle the empty list, function *prod* catches this special case.

Because the *ListPair* functions discard unmatched list elements, there is no checking of matrix dimensions. Adding a 2×5 matrix to a 3×4 matrix yields a 2×4 matrix instead of an exception.

Numerical applications. Before applying the functor, we have to create some structures. We have already seen matrices of real numbers; now it is the integers' turn. Structure *IntZSP* contains just the specified operations specified by *ZSP*:

```
structure IntZSP =
  struct
  type t   = int;
  val zero = 0;
  fun sum  (x,y)  = x+y: t;
  fun prod (x,y)  = x*y: t;
  end;
> structure IntZSP :
>   sig
>   eqtype t
>   val prod : int * int -> t
>   val sum  : int * int -> t
>   val zero : int
>   end
```

Applying the functor to *IntZSP* builds a structure for arithmetic on integer matrices. Two examples are the sum $\left(\begin{smallmatrix}1&2\\3&4\end{smallmatrix}\right) + \left(\begin{smallmatrix}5&6\\7&8\end{smallmatrix}\right) = \left(\begin{smallmatrix}6&8\\10&12\end{smallmatrix}\right)$ and the product $\left(\begin{smallmatrix}1&2\\3&4\end{smallmatrix}\right) \times \left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right) = \left(\begin{smallmatrix}2&1\\4&3\end{smallmatrix}\right)$.

```
structure IntMatrix = MatrixZSP (IntZSP);
> structure IntMatrix : ZSP
IntMatrix.sum ([[1,2],[3,4]], [[5,6],[7,8]]);
> [[6, 8], [10, 12]] : IntMatrix.t
IntMatrix.prod ([[1,2],[3,4]], [[0,1],[1,0]]);
> [[2, 1], [4, 3]] : IntMatrix.t
```

The structure *Complex*, declared in Section 2.21, has several components not specified in *ZSP*. But signature matching ignores surplus components, so we may supply the structure as an argument to $MatrixZSP$. The result is a structure for arithmetic on matrices of complex numbers.

```
structure ComplexMatrix = MatrixZSP (Complex);
> structure ComplexMatrix : ZSP
```

This ability to use one structure for several purposes is a powerful tool for keeping programs simple. It requires, above all, careful design of signatures. Consistent naming conventions help ensure that different modules fit together.

Graph applications. The components $zero$, sum and $prod$ do not have to be the obvious numerical interpretations. Many graph algorithms operate on matrices under surprising interpretations of 0, $+$ and $\times$.

A directed graph consisting of n nodes can be represented by an $n \times n$ ***adjacency matrix***. The (i, j) element of the matrix is a boolean value, indicating the absence or presence of an edge from node i to node j. Typical matrix operations interpret $zero$ as *false*, while sum is disjunction and $prod$ is conjunction.

```
structure BoolZSP =
  struct
  type t   = bool;
  val zero = false;
  fun sum  (x,y) = x orelse y;
  fun prod (x,y) = x andalso y;
  end;
> structure BoolZSP :
>   sig
>   eqtype t
>   val prod : bool * bool -> bool
>   val sum  : bool * bool -> bool
>   val zero : bool
>   end
```

If A is a boolean adjacency matrix, then $A \times A$ represents the graph having an edge from i to j precisely if there is a path of length two from i to j in the graph given by A. Matrix arithmetic can compute the transitive closure of a graph. However, bitwise operations (available in the standard library structure $Word8$) can perform such computations much faster, so let us turn to a more unusual example.

Define $zero$ by infinity (∞), sum by minimum (min) and $prod$ by sum ($+$). The other operations are extended to handle ∞ by $min(\infty, x) = min(x, \infty) = x$ and $\infty + x = x + \infty = \infty$. Thus the triple ($\infty$, min, $+$) satisfies more or less the same laws as (0, $+$, $\times$). But what is this strange arithmetic good for?

Consider a directed graph whose edges are labelled with numbers, indicating the cost (possibly negative!) of travelling along that edge. The corresponding adjacency matrix has numeric elements. Element (i, j) is the cost of the edge from i to j, or infinity if no such edge exists. Let A be an adjacency matrix, and use the strange arithmetic to compute $A \times A$. The (i, j) element of the product is the minimum cost of the paths of length two from i to j. We have the necessary machinery to express a standard algorithm for computing the shortest paths between all nodes of a graph.

Here is a structure implementing the strange arithmetic. It is based on type *int*. It declares *zero* to be not infinity but some large integer.[1] It declares *sum* to be the standard library's minimum function and *prod* to be an extended version of addition.

```
structure PathZSP =
  struct
  type t = int;
  val SOME zero = Int.maxInt;
  val sum       = Int.min
  fun prod(m,n) = if m=zero orelse n=zero then zero
                  else m+n;
  end;
```

Applying our functor to this structure yields a structure for strange arithmetic over matrices. The 'all-pairs shortest paths' algorithm can be coded in a few lines:

```
structure PathMatrix = MatrixZSP (PathZSP);
> structure PathMatrix : ZSP

fun fast_paths mat =
  let val n = length mat
      fun f (m,mat) = if n-1 <= m then mat
                      else f(2*m, PathMatrix.prod(mat,mat))
  in  f (1, mat)  end;
> val fast_paths = fn : PathMatrix.t -> PathMatrix.t
```

Cormen *et al.* (1990) discuss this algorithm (Section 26.1). Let us try it on one of their worked examples. Given the adjacency matrix for a graph of five nodes, *fast_paths* returns the expected result:

```
val zz = PathZSP.zero;
> 1073741823 : int
```

[1] Component *maxInt* of standard library structure *Int* is either *SOME n*, where n is the maximum representable integer, or *NONE*. Any integer exceeding the sum of the absolute values of the edge labels could be used.

```
fast_paths [[0,  3,  8,  zz,  ~4],
            [zz, 0,  zz, 1,   7],
            [zz, 4,  0,  zz,  zz],
            [2,  zz, ~5, 0,   zz],
            [zz, zz, zz, 6,   0]];
> [[0,  1, ~3,  2, ~4],
>  [3,  0, ~4,  1, ~1],
>  [7,  4,  0,  5,  3],
>  [2, ~1, ~5,  0, ~2],
>  [8,  5,  1,  6,  0]] : PathMatrix.t
```

The argument of functor *MatrixZSP* is a structure consisting of only four components. Even smaller structures can be of use, as the next section shows.

An algebraic view. Cormen *et al.* (1990) proceed to put the strange arithmetic on a sound foundation. They define (Section 26.4) the notion of ***closed semiring*** and describe its connection with path algorithms. A closed semiring involves operators analogous to 0, 1, $+$ and $\times$ that satisfy a collection of algebraic laws: $+$ and $\times$ should be commutative and associative, etc. A signature for closed semirings would need to augment *ZSP* with an additional component, *one*. ML modules are ideal for putting such abstractions to use.

Exercise 7.15 Declare a version of *PathZSP* that represents ∞ by a special value, not equal to any integer. Such a structure is appropriate for ML systems such as Poly/ML, where type *int* has no largest value.

Exercise 7.16 Matrices do not have to be lists of lists. Study the standard library structure *Vector*, then write a functor *VMatrixZSP* that represents matrices by vectors of vectors.

7.10 *Generic dictionaries and priority queues*

In Chapter 4 we implemented binary search trees for strings and priority queues for real numbers. Using functors we can lift the type restrictions, generalizing both data structures to arbitrary ordered types. The type and its ordering function will be packaged as a two-component structure.

Sorting can similarly be generalized — without using functors. Simply pass the ordering function as an argument, expressing sorting as a higher-order function. But this is only possible because sorting is an all-in-one operation. Turning the priority queue operations into higher-order functions would permit blunders such as adding items by one ordering and removing them by another.

Ordered types as structures. A mathematician defines an ordered set as a pair $(A, <)$, where A is a set and $<$ is a relation on A that is transitive and so forth.

ML modules can express such mathematical concepts, although the notation is more cumbersome. The signature *ORDER* specifies a type t and an ordering function $compare$:

```
signature ORDER =
  sig
  type t
  val compare: t*t -> order
  end;
```

Recall that the ML library declares $order$ as an enumeration type with constructors *LESS*, *EQUAL* and *GREATER*. The library structures such as $String$, Int and $Real$ have a component $compare$ that takes two operands of the corresponding type. For example, let us package up the string ordering:

```
structure StringOrder: ORDER =
  struct
  type t = string;
  val compare = String.compare
  end;
> structure StringOrder : ORDER
```

We may define our own ordering functions, but note that binary search trees need the ordering to be ***linear***. An ordering $<$ is linear if for all x and y either $x < y$, $x = y$, or $x > y$. Here, it means that if the result of the comparison is *EQUAL* then the two operands really are equal. For priority queues, we could use a partial ordering: if two items are reported as *EQUAL* it means they have equal priority, even if the items themselves are different. (But see Exercise 7.23 below.)

A functor for dictionaries. Section 4.14 outlined the dictionary operations by declaring signature *DICTIONARY*, and implemented it using binary search trees. The implementation was flawed in two respects: keys were artificially restricted to type $string$ and the tree representation was visible outside the structure.

Our new implementation (Figure 7.2) rectifies the first flaw by taking the ordering structure as a parameter, and the second flaw by means of an `abstype` declaration. It forbids equality testing because different binary search trees can represent the same dictionary.

The functor heading tells us that the only operation available for keys is comparison. The functor body resembles the previous, flawed structure. However, it compares keys using its parameter $Key.compare$, instead of $String.compare$. And it declares type key to be $Key.t$ where the old structure declared it to be $string$.

Figure 7.2 *A functor for dictionaries as binary search trees*

```
functor Dictionary (Key: ORDER) : DICTIONARY =
  struct

  type key = Key.t;

  abstype 'a t = Leaf
             | Bran of key * 'a * 'a t * 'a t
    with

    exception E of key;

    val empty = Leaf;

    fun lookup (Leaf, b)              = raise E b
      | lookup (Bran(a,x,t1,t2), b) =
          (case Key.compare(a,b) of
               GREATER => lookup(t1, b)
             | EQUAL   => x
             | LESS    => lookup(t2, b));

    fun insert (Leaf, b, y)              = Bran(b, y, Leaf, Leaf)
      | insert (Bran(a,x,t1,t2), b, y) =
          (case Key.compare(a,b) of
               GREATER => Bran(a, x, insert(t1,b,y), t2)
             | EQUAL   => raise E b
             | LESS    => Bran(a, x, t1, insert(t2,b,y)));

    fun update (Leaf, b, y)              = Bran(b, y, Leaf, Leaf)
      | update (Bran(a,x,t1,t2), b, y) =
          (case Key.compare(a,b) of
               GREATER => Bran(a, x, update(t1,b,y), t2)
             | EQUAL   => Bran(a, y, t1, t2)
             | LESS    => Bran(a, x, t1, update(t2,b,y)));

    end

  end;
```

Applying functor *Dictionary* to the structure *StringOrder* creates a structure of dictionaries with strings for keys.

```
structure StringDict = Dictionary (StringOrder);
> structure StringDict : DICTIONARY
```

Dictionaries can be created and searched. Here, an infix operator eliminates awkward nested calls to *update*:

```
infix |> ;
fun (d |> (k,x)) = StringDict.update(d,k,x);

val dict = StringDict.empty
              |> ("Crecy",1346)
              |> ("Poitiers",1356)
              |> ("Agincourt",1415)
              |> ("Trafalgar",1805)
              |> ("Waterloo",1815);
> val dict = - : int StringDict.t
StringDict.lookup(dict,"Poitiers");
> 1356 : int
```

Priority queues: an example of a substructure. Section 4.16 outlined the priority queue operations by declaring signature *PRIORITY_QUEUE*, and implemented it using binary trees. The implementation had the same two flaws as that of dictionaries. Instead of covering the same ideas again, let us examine something new: substructures.

One difference between dictionaries and priority queues is the rôle of the ordering. The dictionary functor takes an ordering because it uses search trees; alternative implementations might take an equality test or a hashing function. But a priority queue is intrinsically concerned with an ordering: having accumulated items, it returns the smallest item first. So let us modify the result signature to make the ordering explicit:

```
signature PRIORITY_QUEUE =
  sig
  structure Item : ORDER
  type t
  val empty      : t
  val null       : t -> bool
  val insert     : Item.t * t -> t
  val min        : t -> Item.t
  val delmin     : t -> t
  val fromList   : Item.t list -> t
  val toList     : t -> Item.t list
  val sort       : Item.t list -> Item.t list
  end;
```

Signature *PRIORITY_QUEUE* specifies a substructure *Item* matching signature *ORDER*. The type of items is *Item.t*, while the type of priority queues is simply t. Thus *min*, which returns the smallest item in a queue, has type $t \to Item.t$.

Every priority queue structure carries the ordering with it. If *PQueue* is an instance of the signature, we may compare x with y by writing

```
PQueue.Item.compare(x,y)
```

Under this approach, system components are specified as substructures. The previous version of *PRIORITY_QUEUE*, which many people prefer for its simplicity, specified a type *item* instead of a structure *Item*.

The corresponding functor has the following outline. Most of the body is omitted; it is similar to Figure 4.4 on page 163.

```
functor PriorityQueue (Item: ORDER) : PRIORITY_QUEUE =
  struct
  structure Item = Item;

  fun x <= y = (Item.compare(x,y) <> GREATER);

  abstype t = ...
    with
    ⋮
    end

  end;
```

The structure declaration of *Item* may seem to do nothing, because *Item* is already visible in the functor body. But the result signature requires this declaration. It is analogous to the many type declarations we have seen in structures and functors. Nested structure declarations do not have to be trivial; all the forms valid at top level are also valid inside another structure.

The functor redeclares the infix operator <= to denote 'less than or equal' on items. In Chapter 4, binary search trees used *compare* for their ordering, while priority queues used <=. It would be silly to declare distinct versions of signature *ORDER* for the two functors, or to specify all the different relational operators. Simple, uniform interfaces let modules fit together easily.

The `abstype` declaration can declare fresh tree constructors, as in *Dictionary*. Or it can use the existing constructors *Lf* and *Br* (declared at top level in Section 4.13) by declaring a dummy constructor, as in type *queue*1 above.

Exercise 7.17 Write a new version of functor *Dictionary*, representing a dictionary by a list of (*key*, *item*) pairs ordered by the keys.

Exercise 7.18 Complete the `abstype` declaration above, trying both alternatives. Which one do you prefer?

Exercise 7.19 Write a new version of functor *PriorityQueue*, representing a priority queue by an increasing list instead of a binary tree.

Exercise 7.20 Write a functor *Sorting* whose argument is an instance of signature *ORDER* and whose result implements both quick sort and merge sort. What is the point of providing more than one sorting algorithm?

Building large systems using modules

Through numerous small examples we have surveyed the basic features of the modules language. We have seen a variety of uses of structures:

- The library structure *List* holds related declarations, but more can be declared in terms of the list constructors.
- Structure *AbsQueue*3 exports an abstract type together with all its primitive operations. Further queue operations can be expressed only in terms of those.
- The *ZSP* structures serve as the arguments or results of a functor. They have only a few components, namely those operations that are pertinent to the functor.

A large system ought be organized into hundreds of small structures such as those above. The organization should be hierarchical: major subsystems should be implemented as structures whose components are structures of the layer below. More chaotic programmers may find themselves presiding over a few huge structures, each consisting of hundreds or thousands of components.

A well-organized system will have many small signatures. Component specifications will obey strict naming conventions. In a group project, team members will have to agree upon each signature. Subsequent changes to signatures must be controlled rigorously.

The system will include some functors, possibly many. If the major subsystems are implemented independently, they will all have to be functors.

The modules language contains constructs, many of them obscure, that make all these working practices possible. So let us take a closer look at modules.

7.11 *Functors with multiple arguments*

An ML function takes only one argument. Multiple arguments are usually packaged as a tuple. Alternatively, they can be packaged as a record.

Higher-order functions can express multiple arguments through the device of currying.

A functor also takes only one argument. Multiple arguments are packaged as a structure, which is analogous to passing a function's arguments as a record. The syntax is clumsy but workable. Some compilers extend Standard ML by providing higher-order functors, which allow currying.

A functor for lexicographic orderings. Our first example is a two-argument functor. If $<_\alpha$ is an ordering on type α and $<_\beta$ is an ordering on type β then the ***lexicographic ordering*** $<_{\alpha\times\beta}$ on type $\alpha \times \beta$ is defined by

$$(a', b') <_{\alpha\times\beta} (a, b) \text{ if and only if } a' <_\alpha a \text{ or } (a' = a \text{ and } b' <_\beta b).$$

The functor *LexOrder* has result signature *ORDER*. It takes two formal parameters: the structures $O1$ and $O2$, also of signature *ORDER*. Its declaration illustrates ML's general syntax for functor headings:

```
functor LexOrder (structure O1: ORDER
                  structure O2: ORDER) : ORDER =
  struct
  type t = O1.t * O2.t;
  fun compare ((x1,y1), (x2,y2)) =
      (case O1.compare (x1,x2) of
           EQUAL => O2.compare (y1,y2)
         | ord   => ord)
  end;
```

The formal parameter list is simply a signature specification — a signature, but without the `sig` and `end` brackets. The specified components are visible in the functor body. The functor may be applied to any structure matching the specification: any structure containing two substructures $O1$ and $O2$ that match signature *ORDER*. The structure can be given by any structure expression, including another functor application.

Structure *StringOrder* has been declared above, and *IntegerOrder* can be declared similarly. We can supply those two arguments to the functor like this:

```
structure StringIntOrd = LexOrder(structure O1=StringOrder
                                  structure O2=IntegerOrder);
> structure StringIntOrd : ORDER
```

An argument consisting of a list of declarations is regarded as a structure expression. The multiple arguments form the body of the structure, and we may omit the `struct` and `end` brackets.

A demonstration will remind us of the functor's purpose. Combining the

Figure 7.3 *A dictionary functor using association lists*

```
functor AssocList (eqtype key) : DICTIONARY =
  struct
  type key  = key;
  type 'a t  = (key * 'a) list;

  exception E of key;

  val empty = [];

  fun lookup ((a,x)::pairs, b) = if a=b then x
                                         else lookup(pairs, b)
    | lookup ([], b)           = raise E b;

  fun insert ((a,x)::pairs, b, y) = if a=b then raise E b
                                            else (a,x)::insert(pairs, b, y)
    | insert ([], b, y)           = [(b,y)];

  fun update (pairs, b, y) = (b,y)::pairs;

  end;
```

orderings on strings and integers yields an ordering on (string, integer) pairs. The ordering on strings takes precedence over that on integers.

```
StringIntOrd.compare (("Edward", 3), ("Henry", 2));
> LESS : order
StringIntOrd.compare (("Henry", 6), ("Henry", 6));
> EQUAL : order
StringIntOrd.compare (("Henry", 6), ("Henry", 5));
> GREATER : order
```

Association lists; the `eqtype` *specification.* ML's functor syntax for multiple arguments does not require those arguments to be structures. They can be anything that a signature can specify, including types, values and exceptions.

The following example demonstrates the `eqtype` specification as well as the general functor syntax. We have previously implemented dictionaries as binary search trees. Lists of pairs are a simpler but slower representation. As in Section 3.16, the lookup operation compares keys using equality.

An `eqtype` specification may appear in any signature. It specifies types that admit equality. A structure only matches the signature if it declares actual types that really do admit equality. Within a functor body, equality testing is permitted on types specified by `eqtype`.

The functor's formal parameter list is a signature specification (Figure 7.3). It specifies one argument, an equality type. The general functor syntax lets us view *AssocList* as a functor whose formal parameter is a type. Because type *key* is specified as an `eqtype`, it admits equality testing within *AssocList*. Here are two functor applications:

```
structure StringIntAList = AssocList (type key = string*int);
> structure StringIntAList : DICTIONARY
structure FunctionAList = AssocList (type key = int->int);
> Error: type key must be an equality type
```

We may apply the functor to $string \times int$ because this type admits equality. The type $int \to int$ is rejected.

Functors with no arguments. The ***empty structure*** consists of no components:

```
struct end
```

Its signature is the ***empty signature***:

```
sig end
```

The empty structure is mainly used as the argument of a functor. There it is analogous to the empty tuple (), which is mainly used when a function does not depend on the value of its argument. Recall the use of a function to represent the tail of a sequence (Section 5.12). Null arguments are also used with imperative programming. Our functor example involves references, which are discussed in Chapter 8.

Functor *MakeCell* takes a null argument. Its empty formal parameter list constitutes an empty signature. Every time *MakeCell* is called, it allocates a fresh reference cell and returns it as part of a structure. The cell initially contains 0:

```
functor MakeCell () =  struct  val cell = ref 0  end;
> functor MakeCell : <sig>
```

Here are two functor invocations. The empty actual parameter list constitutes the body of an empty structure.

```
structure C1 = MakeCell ()
and       C2 = MakeCell ();
> structure C1 : sig val cell : int ref end
> structure C2 : sig val cell : int ref end
```

Structures $C1$ and $C2$ have been created in the same way, but they contain distinct reference cells. Let us store a 1 in $C1$'s cell, then inspect both of them:

```
C1.cell := 1;
> () : unit
C1.cell;
> ref 1 : int ref
C2.cell;
> ref 0 : int ref
```

The cells hold different integers. Because *MakeCell* is a functor, and not just a structure, it can allocate as many distinct cells as required.

Functor syntax confusion. The general functor syntax, with a signature specification in the functor heading, handles any number of arguments. But what if we have exactly one argument, a structure? We could use the primitive functor syntax; it is more concise and direct than the general syntax, which creates another structure. On the other hand, using both syntaxes in a program may lead to confusion. All our early examples used the primitive syntax:

```
functor TestQueue  (Q: QUEUE) ...
```

A different programmer might have used the general syntax:

```
functor TestQueue2 (structure Q: QUEUE) ...
```

These declarations differ only by the keyword `structure` in the formal parameter list, which might be overlooked. To avoid an error message, each functor should be invoked with the corresponding argument syntax:

```
TestQueue  (Queue3)
TestQueue2 (structure Q = Queue3)
```

For uniformity's sake, some programmers prefer to use the general syntax exclusively.

Exercise 7.21 Write a version of *AssocList* that does not involve `eqtype`. Instead, it should employ a signature similar to *ORDER*.

Exercise 7.22 Functor *AssocList* does not hide the representation of dictionaries; write a version that declares an abstract type.

Exercise 7.23 In a partial ordering, some pairs of elements may be unrelated. Signifying this outcome by *EQUAL* is not satisfactory in general; it would give the wrong results for the definition of lexicographic ordering. John Reppy suggests representing outcomes of comparisons by values of type *order option*, using *NONE* to signify 'unrelated.' Declare the signature *PORDER* for partial orderings, and the functor *LexPOrder* for combining partial orderings, by analogy with *ORDER* and *LexOrder*.

Exercise 7.24 (Continuing the previous exercise.) If α is a type and $<_\beta$ is a partial ordering on type β and f is a function of type $\alpha \to \beta$ then we can define a partial ordering $<$ over type α by $x' < x$ if and only if $f(x') <_\beta f(x)$. (Note that $f(x') = f(x)$ need not imply $x' = x$.) Declare a three-argument functor that implements this definition.

Exercise 7.25 Which structures are instances of the empty signature? In other words, which structures are legal arguments to functor *MakeCell*?

7.12 *Sharing constraints*

When modules are combined to form a larger one, special care may be needed to ensure that the components fit together. Consider the problem of combining dictionaries and priority queues, ensuring that their types agree.

Above we applied the functor *Dictionary* to the argument *StringOrder*, creating the structure *StringDict*. We then declared *dict* to be a dictionary indexed by strings. We can similarly apply *PriorityQueue* to *StringOrder*, creating a structure for priority queues of strings.

```
structure StringPQueue = PriorityQueue (StringOrder);
> structure StringPQueue : PRIORITY_QUEUE
```

Let us now declare *pq* to be a priority queue of strings:

```
StringPQueue.insert("Agincourt", StringPQueue.empty);
> - : StringPQueue.t
StringPQueue.insert("Crecy", it);
> - : StringPQueue.t
val pq = StringPQueue.insert("Poitiers", it);
> val pq = - : StringPQueue.t
```

Since elements of *pq* are strings, and *dict* is indexed by strings, the least element of *pq* may serve as a search key into *dict*.

```
StringDict.lookup(dict, StringPQueue.min pq);
> 1356 : int
```

We have used dictionaries and priority queues together, but only for type *string*. Generalizing this expression to an arbitrary ordered type requires a functor. In the functor body, the expression has the form

```
Dict.lookup(dict, PQueue.min pq)
```

where *PQueue* and *Dict* are structures matching signatures *PRIORITY_QUEUE* and *DICTIONARY*, respectively. But do the types agree?

$$PQueue.min : PQueue.t \to PQueue.Item.t$$
$$Dict.lookup : \alpha\ Dict.t \times Dict.key \to \alpha$$

The call of *Dict.lookup* is permissible only if *PQueue.Item.t* is the same type as *Dict.key*. One way to ensure this is for the functor to build the structures *PQueue* and *Dict* itself. The following functor takes an ordered type as an argument, and supplies it to functors *PriorityQueue* and *Dictionary*. Our expression appears as the body of function *lookmin*:

```
functor Join1 (Order: ORDER) =
  struct
  structure PQueue = PriorityQueue (Order);
  structure Dict   = Dictionary    (Order);

  fun lookmin(dict, pq) = Dict.lookup(dict, PQueue.min pq);

  end;
```

It is often useful for one functor to call another. But functor *Join*1 does not combine existing structures: it makes new ones. This approach could create many duplicate structures.

Our functor should take existing structures *PQueue* and *Dict*, checking that their types are compatible. A ***sharing constraint*** can compel types to agree:

```
functor Join2 (structure PQueue : PRIORITY_QUEUE
               structure Dict   : DICTIONARY
               sharing type PQueue.Item.t = Dict.key) =
  struct
  fun lookmin(dict, pq) = Dict.lookup(dict, PQueue.min pq);
  end;
```

We have reverted to the multiple-argument functor syntax; sharing constraints are a form of signature specification. In the body of the functor, the constraint guarantees that the two types are identical. The type checker therefore accepts the declaration of *lookmin*. When the functor is applied to actual structures, the ML compiler insists that the two types really are the same.

To demonstrate the functor, we shall need priority queues and dictionaries of integers:

```
structure IntegerPQueue = PriorityQueue (IntegerOrder);
> structure IntegerPQueue : PRIORITY_QUEUE
structure IntegerDict = Dictionary (IntegerOrder);
> structure IntegerDict : DICTIONARY
```

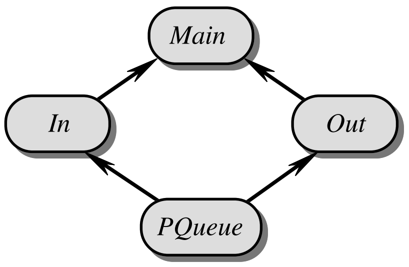
Two string-based structures can be combined, and so can two integer-based structures. In each case, function *lookmin* takes a dictionary and a priority queue based on the same type.

```
structure StringCom = Join2 (structure PQueue = StringPQueue
                             structure Dict   = StringDict);
> structure StringCom
> : sig
>   val lookmin: 'a StringDict.t * StringPQueue.t -> 'a
>   end
structure IntegerCom = Join2 (structure PQueue = IntegerPQueue
                              structure Dict   = IntegerDict);
> structure IntegerCom
> : sig
>   val lookmin: 'a IntegerDict.t * IntegerPQueue.t -> 'a
>   end
```

But if we try to mix the types, the compiler rejects the declaration:

```
structure Bad = Join2 (structure PQueue = IntegerPQueue
                       structure Dict   = StringDict);
> Error: type sharing violation
>   StringDict.key # IntegerPQueue.Item.t
```

Sharing constraints on structures. When functors combine system components, common substructures may need sharing constraints. Here is a sketch of a typical situation. Structure *In* inputs problems; structure *Out* outputs solutions. The two components communicate via a priority queue of goals, in structure *PQueue*. Structure *Main* coordinates the program via *In* and *Out*.



Suppose that *In* and *Out* match the following signatures:

```
signature IN =
  sig
  structure PQueue: PRIORITY_QUEUE
  type problem
  val  goals: problem -> PQueue.t
  end;
signature OUT =
    sig
    structure PQueue: PRIORITY_QUEUE
    type solution
    val  solve: PQueue.t -> solution
    end;
```

A functor to combine *In* and *Out* might look like this:

```
functor MainFunctor (structure In: IN and Out: OUT
                     sharing In.PQueue = Out.PQueue) =
  struct
  fun tackle(p) = Out.solve(In.goals p)
  end;
```

Because the structures *In*.*PQueue* and *Out*.*PQueue* are declared as sharing, the types *In*.*PQueue*.*t* and *Out*.*PQueue*.*t* are identical in the functor body. (Observe the use of `and` to specify two structures concisely.)

When building the system, put the same structure *PQueue* into *In* and *Out*. The functor *MainFunctor* will then accept *In* and *Out* as arguments, since they will satisfy the sharing constraint.

Understanding sharing constraints. Sharing is one of the most difficult aspects of ML modules. Although sharing constraints may appear in any signature, they are only necessary if the signature specifies a functor argument. The more functors you use, the more sharing constraints you will need.

A type error is the usual warning that a sharing constraint might be necessary. In our previous example, omitting the constraint might cause the error 'type conflict: expected *In*.*PQueue*.*t*, found *Out*.*PQueue*.*t*.' Unfortunately, some compilers produce cryptic error messages.

The type error could be eliminated by imposing a sharing constraint on those types:

```
sharing type In.PQueue.t = Out.PQueue.t
```

The structure sharing constraint actually used in *MainFunctor* is stronger: it implies type sharing all the way down. It implies that the types *In*.*PQueue*.*Item*.*t* and *Out*.*PQueue*.*Item*.*t* are also shared.

ML enforces sharing constraints by comparing the types' identities. Each new datatype or abstract type is regarded as distinct from all previously existing types.

```
structure DT1 = struct datatype t = C end;
structure DT2 = struct datatype t = C end;
structure DT3 = struct type t = DT1.t end;
```

The types $DT1$.t and $DT2$.t are distinct, even though they arise from identical `datatype` declarations. Type abbreviations preserve the identity, so types $DT1$.t and $DT3$.t are the same.

Exercise 7.26 Explain ML's response to the following declarations.

```
signature TYPE = sig type t end;
functor Funny (structure A: TYPE and B: TYPE
               sharing A=B) = A;
structure S1 = Funny (structure A=DT1 and B=DT1);
structure S2 = Funny (structure A=DT2 and B=DT2);
structure S3 = Funny (structure A=S1  and B=S2);
```

Exercise 7.27 Suppose that the functors *Input* and *Output* are declared as follows:

```
functor Input (structure PQueue: PRIORITY_QUEUE): IN =
  struct
  structure PQueue = PQueue;
  fun goals ...;
  end;
functor Output (structure PQueue: PRIORITY_QUEUE): OUT =
  struct
  structure PQueue = PQueue;
  fun solve ...;
  end;
```

By applying these functors, declare structures that may be given to $MainFunctor$. Then declare structures that have the required signatures but violate the functor's sharing constraint.

Exercise 7.28 The functors *Input* and *Output* declared above incorporate the formal parameter $PQueue$ into the result structure. Modify them to generate a fresh instance of *PRIORITY_QUEUE* instead. How will this affect $MainFunctor$?

7.13 *Fully-functorial programming*

It is a truism that one should never declare a procedure that is called only once. We have never declared a functor to be called only once. Each formal parameter has had a choice of actual parameters; for example, the parameter $Order$ could be instantiated by $StringOrder$ or $IntegerOrder$. Non-generic program units have been coded as structures, not as functors.

But declaring procedures is now regarded as good style, even if they are called only once. There are good reasons for declaring more functors than are strictly necessary. Some programmers code almost entirely with functors, writing structures only to supply as arguments to functors. Their functors and signatures are self-contained: they refer only to other signatures and to components of the standard library.

If all program units are coded as functors then they can be written and compiled separately. First, the signatures are declared; then, the functors are coded. When a functor is compiled, error messages may reveal mistakes and omissions in the signatures. Revised signatures can be checked by recompiling the functors.

The functors may be coded in any order. Each functor refers to signatures, but not to structures or other functors. Some people prefer to code from the top down, others from the bottom up. Several programmers can code their functors independently.

Once all the functors have been written and compiled, applying them generates a structure for each program unit. The final structure contains the executable program. A functor can be modified, recompiled and a new system built, without recompiling the other functors, provided no signatures have changed. Applying the functors amounts to linking the program units. Different configurations of a system can be built.

Functors for binary trees. From Section 4.13 onwards we declared structures for binary trees, flexible arrays, etc. We even declared *tree* as a top level datatype. The fully-functorial style requires every program unit to be specified by a self-contained signature.

We must now declare a signature for binary trees. The signature must specify the datatype *tree*, as it will not be declared at top level.

```
signature TREE =
  sig
  datatype 'a tree = Lf  |  Br of 'a * 'a tree * 'a tree
  val size    : 'a tree -> int
  val depth   : 'a tree -> int
  val reflect : 'a tree -> 'a tree
  ⋮
  end;
```

We must declare a signature for the Braun array operations. The signature specifies *Tree* as a substructure to provide access to type *tree*.[2]

[2] It could instead specify type *tree* directly; see Section 7.15.

```
signature BRAUN =
  sig
  structure Tree: TREE
  val sub    : 'a Tree.tree * int -> 'a
  val update : 'a Tree.tree * int * 'a -> 'a Tree.tree
  val delete : 'a Tree.tree * int -> 'a Tree.tree
  ⋮
  end;
```

Signature *FLEXARRAY* (Section 4.15) is self-contained, as it depends only on the standard type *int*. Signatures *ORDER* and *PRIORITY_QUEUE* (Section 7.10) are also self-contained. Since a signature may refer to others, the declarations must be made in a correct order: *TREE* must be declared before *BRAUN*, and *ORDER* before *PRIORITY_QUEUE*.

Since our functors do not refer to each other, they can be declared in any order. The functor *PriorityQueue* can be declared now, even if its implementation relies on binary trees. The functor is self-contained: it takes a binary tree structure as a formal parameter, *Tree*, and uses it for access to the tree operations:

```
functor PriorityQueue (structure Order : ORDER
                       structure Tree  : TREE)
        : PRIORITY_QUEUE =
    ⋮
    abstype t = PQ of Item.t Tree.tree
    ⋮
```

The structure *Flex* (see page 158) can be turned into a functor *FlexArray* taking *Braun* as its formal parameter. The body of the functor resembles the original structure declaration. But the tree operations are now components of the substructure *Braun*.*Tree*.

```
functor FlexArray (Braun: BRAUN) : FLEXARRAY =
    ⋮
    val empty = Array(Braun.Tree.Lf,0);
    ⋮
```

The structure *Braun* can similarly be turned into a functor *BraunFunctor* taking *Tree* as its formal parameter.

```
functor BraunFunctor (Tree: TREE) : BRAUN = ...
```

Even structure *Tree* can be made into a functor: taking the null parameter.

```
functor TreeFunctor () : TREE = struct ... end;
```

Now all the functors have been declared.

Linking the functors together. The final phase, after all the code has been written, is to apply the functors. Each structure is built by applying a functor to previously created structures. To begin, applying *TreeFunctor* to the empty argument list generates the structure *Tree*.

```
structure Tree = TreeFunctor ();
> structure Tree : TREE
```

Functor applications create the structures *Braun* and *Flex*:

```
structure Braun = BraunFunctor (Tree);
structure Flex  = FlexArray (Braun);
```

The structure *StringOrder* is declared as it was above:

```
structure StringOrder = ...;
```

Now structure *StringPQ* can be declared, as before, by functor application:

```
structure StringPQueue =
   PriorityQueue (structure Item = StringOrder
                  structure Tree = Tree);
```

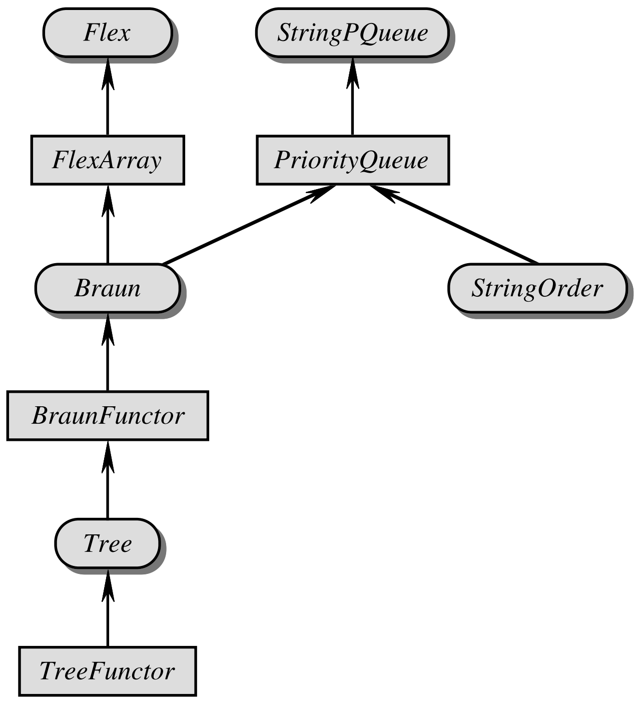
Figure 7.4 portrays the complete system, with structures as rounded boxes and functors as rectangles. Most of the structures were created by functors; only *StringOrder* was written directly.

The drawbacks of the fully-functorial style should be evident. All the functor declarations clutter the code; even inventing names for them is hard. Sharing constraints multiply. If we continue this example, we can expect many constraints to ensure that structures share the same *Tree* substructure. With some ML systems, your compiled code may be twice as large as it should be, because it exists both as functors and as structures.

A good compromise is to use functors for all major program units: those that must be coded independently. Many of them will be generic anyway. Lower-level units can be declared as structures. Biagioni *et al.* (1994) have organized the layers of a large networking system using signatures and functors; components can be joined in various ways to meet specialized requirements.

When is a signature self-contained? Names introduced by a specification become visible in the rest of the signature. Signature *TREE* specifies the type *tree* and then uses it, and type *int*, to specify the type of *size*. Predefined names like *int* and the standard library structures are said to be ***pervasive***: they are visible

Figure 7.4 *Structures and functors involving trees*



everywhere. A name that occurs in a signature and that has not been specified there is said to be ***free*** in that signature. The only name occurring free in *TREE* is *int*.

David MacQueen wrote the original proposal for Standard ML modules (in Harper *et al.* (1986)). He proposed that signatures could not refer to names declared elsewhere in a program, except names of other signatures. A signature could refer to structures specified within the same signature, but not to free-standing structures. Thus, every signature was completely self-contained, and every structure carried with it the structures and types it depended upon. This restriction, the ***signature closure rule***, was eventually relaxed to give programmers greater freedom.

In the fully-functorial style, the structures are declared last of all. Signatures will naturally obey the signature closure rule, as there will be no structures for them to refer to. The only names free in *BRAUN* are the pervasive type *int* and the signature *TREE*. If the line

```
structure Tree: TREE
```

were removed, the signature would depend on some structure *Tree* already declared, since it mentions *Tree*.*tree* in the types of the functions. This is an acceptable programming style, but it is not the fully-functorial style, and it violates the signature closure rule.

Functors and signature constraints. In the fully-functorial style of programming, each functor refers to structures only as formal parameters. Nothing is known about an argument structure except its signature, just as when an actual structure is given an opaque signature constraint.

Suppose a functor's formal parameters include a structure *Dict* of signature *DICTIONARY*. In the functor body, type α *Dict*.*t* behaves like an abstract type: it has no constructors to allow pattern-matching, and equality testing is forbidden. Type *Dict*.*key* is similarly abstract; unless a sharing constraint equates it with some other type, we shall be unable to call *Dict*.*lookup*.

Exercise 7.29 Write a functor with no formal parameters and result signature *QUEUE*, implementing Representation 3 of queues.

Exercise 7.30 Specify a signature *SEQUENCE* for an abstract type of lazy lists, and implement the type by writing a functor with result signature *SEQUENCE*. Write a functor that takes instances of *QUEUE* and *SEQUENCE*, and declares search functions like *depthFirst* and *breadthFirst* (Section 5.17).

Exercise 7.31 List which names appear free in signatures *QUEUE1*, *QUEUE2*, *QUEUE3*, *DICTIONARY* (Section 4.14) and *FLEXARRAY* (Section 4.15).

7.14 *The* `open` *declaration*

Compound names get cumbersome when structures are nested. In the body of functor *FlexArray*, the type of binary trees is called *Braun*.*Tree*.*tree* and its constructors are called *Braun*.*Tree*.*Lf* and *Braun*.*Tree*.*Br*. The type and its constructors behave in the normal manner, but any patterns written using the constructor notation are likely to be unreadable.

Although the fully-functorial style makes the problem worse, long compound names can arise in any large program. Fortunately there are many ways of abbreviating such names.

Opening a structure declares its items so that they are known by their simple names. The syntax of an `open` declaration is

```
open Id
```

where *Id* is the (possibly compound) name of a structure. Only one level of a structure is opened at a time. After declaring

```
open Braun;
```

we may write *Tree* and *Tree*.*Lf* instead of *Braun*.*Tree* and *Braun*.*Tree*.*Lf*. If we go on to declare

```
open Tree;
```

then we may write *Lf* and *Br* instead of *Tree*.*Lf* and *Tree*.*Br*. In the scope of this `open` declaration, *Lf* and *Br* denote constructors and may not be redeclared as values.

Local `open` *declarations.* Since `open` is a declaration, a `let` or `local` construct can restrict its scope. Recall that `let` makes a declaration that is private to an expression, while `local` (Section 2.18) makes a declaration that is private to another declaration.

Here is an example of a local `open` declaration that also demonstrates how `open` can be misused. Functor *FlexArray* might use `local` as follows:

```
functor FlexArray (Braun: BRAUN) : FLEXARRAY =
  struct
    local open Braun Braun.Tree
    in
    datatype 'a array = Array of 'a tree * int;
    val empty = Array(Lf,0);
    fun length (Array(_,n)) = n;
    fun sub    ...
    fun update ...
    fun delete ...
    fun loext  ...
    fun lorem  ...
    end
  end;
```

The `open` declaration makes the components of *Braun* and *Braun*.*Tree* visible, while `local` restricts their scope to the functor body. We no longer need to write compound names.

Or do we? Recall that the functor implements flexible arrays in terms of Braun arrays. Subscripting on flexible arrays uses subscripting on Braun arrays. Both operations are called *sub*; to avoid a name clash we must write a compound name:

```
    fun sub (Array(t,n), k) =
        if 0<=k andalso k<n then Braun.sub(t,k+1)
        else raise Subscript;
```

Omitting the prefix *Braun*. above would create a spurious recursive call to *sub*, and a type error. So opening *Braun* does not accomplish anything. There is no need to open *Braun*.*Tree* either, as the functor body uses this prefix only twice.

Structure expressions using `let`. A better candidate for `open` is *BraunFunctor*, which uses the tree constructors extensively (see Figure 7.5 on the following page). Opening structure *Tree* spares us from writing compound names for the constructors in expressions such as *Br*(*w*, *Lf*, *Lf*).

The functor uses a new `let` construct, one that operates on structures. Suppose the *Str* is a structure expression requiring the declaration *D*. Then evaluating the structure expression

```
let D in Str end
```

yields the result of evaluating *Str*, while delimiting the scope of *D*. If *Str* has the form `struct ...end`, as does the body of *BraunFunctor*, then we can equivalently write (as in the previous example)

```
struct
local D in ...end
end
```

However, we can use `let` with other structure expressions, such as functor applications. It is especially useful when a structure is used more than once:

Figure 7.5 *Example of* `let open` *in a functor body*

```
functor BraunFunctor (Tree: TREE) : BRAUN =
  let open Tree in
    struct
    structure Tree = Tree;

    fun sub (Lf, _)          = raise Subscript
      | sub (Br(v,t1,t2), k) =
          if k = 1 then v
          else if k mod 2 = 0
               then sub (t1, k div 2)
               else sub (t2, k div 2);

    fun update (Lf, k, w)          =
          if k = 1 then Br (w, Lf, Lf)
          else raise Subscript
      | update (Br(v,t1,t2), k, w) =
          if k = 1 then Br (w, t1, t2)
          else if k mod 2 = 0
               then Br (v,  update(t1, k div 2, w),  t2)
               else Br (v,  t1,  update(t2, k div 2, w));

    fun delete (Lf, n)          = raise Subscript
      | delete (Br(v,t1,t2), n) =
          if n = 1 then Lf
          else if n mod 2 = 0
               then Br (v,  delete(t1, n div 2),  t2)
               else Br (v,  t1,  delete(t2, n div 2));

    fun loext (Lf, w)          = Br(w, Lf, Lf)
      | loext (Br(v,t1,t2), w) = Br(w, loext(t2,v), t1);

    fun lorem Lf                            = raise Size
      | lorem (Br(_,Lf,_))                  = Lf
      | lorem (Br(_, t1 as Br(v,_,_), t2)) = Br(v, t2, lorem t1);

    end
  end;
```

```
functor QuadOrder (O: ORDER) : ORDER =
  let structure OO = LexOrder (structure O1 = O
                                structure O2 = O)
  in  LexOrder (structure O1 = OO
                structure O2 = OO)
  end;
```

Functor $QuadOrder$ takes an ordered structure and returns the lexicographic ordering for quadruples of the form $((w, x), (y, z))$. Internally it creates the structure OO, which defines the ordering for pairs.

Infix operators in structures. Infix directives issued inside a structure have no effect outside. When a structure is opened, its names are made visible as ordinary identifiers, not as infix operators; restoring their infix status requires new infix directives. A compound name can never become an infix operator; only simple names are permitted in an infix directive.

Top level infix directives, issued outside any structure, have global scope. Opening a structure can bind or re-bind these top level operators.

Re-binding identifiers using open*.* Opening several structures can declare hundreds of names at a stroke. Unless these names are descriptive, we may not be able to remember which structure they belong to. Using open to override existing bindings can be particularly confusing.

ML systems may provide the library structures *Real*32, *Real*64, *Real*96, etc., which implement the standard floating point operations in various precisions. The structures match signature $REAL$, which specifies a type *real* and operators such as +, −, * and /.

Compound names make these structures hard to use. The 64-bit version of $(a/x + x)/y$ is incomprehensible:

```
Real64./ (Real64.+ (Real64./(a,x), x), y)
```

A local open declaration restores readability:

```
let open Real64 in (a/x + x) / y end
```

Unfortunately, opening *Real*64 redeclares all the numeric operators, obliterating their overloading. We can no longer write integer expressions such as n+1. The integer operations are still accessible via their home structure: we can still write *Int*.+(n,1). But is this an improvement?

Opening *Real*64 at top level is plainly wrong. There is no way of restoring the overloading. It is better to open *Real*64 with a small scope, as above, or to declare new infix operators and bind them to the 64-bit functions.

Alternatives to open*.* As these examples show, open can cause obscurity. Our structures have been designed to take advantage of compound names. The simple names of the items are too short. Compound names like *Braun*.*sub* and

Flex.sub do not merely avoid clashes; they are informative and reinforce our knowledge of the program's organization.

We can shorten compound names by declaring abbreviations, without using `open`. Our declaration of functor *FlexArray* can be improved:

```
functor FlexArray (Braun: BRAUN) : FLEXARRAY =
  struct
    local structure T = Braun.Tree
    in
    datatype 'a array = Array of 'a T.tree * int;
    val empty = Array(T.Lf,0);
    end
    ⋮
  end;
```

Instead of opening *Braun.Tree*, we declare structure *T* to abbreviate it. Having to write *T.tree* is perfectly acceptable, and patterns expressed using *T.Lf* and *T.Br* can be succinct.

Some programmers will regard compound identifiers as unacceptable, at least for heavily used items. But there is no need to open a large module if only a few of the items are really needed. An `open` declaration can be replaced by separate abbreviations:

```
type 'a queue       = 'a Queue.t;
val hd              = Queue.hd;
exception QEmpty    = Queue.E;
```

The last line makes *QEmpty* a synonym for *Queue.E*; it is a constructor and may even appear in exception handlers. In such exception bindings, the right-hand side must be the name of an exception constructor.

Selective use of `open`. In Section 4.13 we declared datatype *tree* at top level to avoid having compound constructor names. That was poor style; every type and variable ought to belong to a home structure. Opening the full tree structure is equally undesirable. And there is no analogue of exception bindings for exporting individual datatype constructors.

We must use `open`, but we can do so selectively. Core declarations — in this case, the datatype *tree* and the function *depth* — can be declared in a substructure:

```
structure Tree =
  struct

  structure Export =
    struct
    datatype 'a tree = Lf
                     | Br of 'a * 'a tree * 'a tree;

    fun depth Lf              = 0
      | depth (Br(v,t1,t2)) = 1 + Int.max (depth t1, depth t2);
    end;

  open Export;

  fun size Lf             = 0
    | size (Br(v,t1,t2)) = 1 + size t1 + size t2;

  ⋮
  end;
```

The substructure *Export* contains the items that are to be exported to top level. It is immediately opened, exporting the items to the main structure. Later we can export the core items, leaving the others accessible only by compound names:

```
open Tree.Export;
depth Lf;
> 0 : int
Tree.size Lf;
> 0 : int
```

A variation on this idea is to declare the structure of core items at top level. It might be called *TreeCore*, with its own signature *TREECORE*. Other structures and signatures for trees could refer to the core ones.

Exercise 7.32 Explain why the declaration of *StrangePQueue* is valid.

```
functor StrangePQueue () =
  let structure UsedTwice = struct open StringOrder Tree end
  in  PriorityQueue (structure Item = UsedTwice
                     structure Tree = UsedTwice)
  end;
```

Exercise 7.33 What is the effect of the following declaration?

```
open Queue3;  open Queue2;
```

Exercise 7.34 What is wrong with the following attempt at multiple-precision arithmetic?

```
functor MultiplePrecision (F: REAL) =
  struct
  fun half x = F./(x, 2.0)
  end;
```

7.15 *Signatures and substructures*

Complex programs require complex signatures. When structures are nested, their signatures can become cluttered with excessively long compound names. Suppose we declare a signature for pairs of Braun arrays, specifying a substructure matching signature *BRAUN*:

```
signature BRAUNPAIR0 =
  sig
  structure Braun: BRAUN
  val zip: 'a Braun.Tree.tree * 'b Braun.Tree.tree ->
          ('a*'b) Braun.Tree.tree
  ⋮
  end;
```

The compound names render the type of *zip* unreadable. As with structures, there are a number of approaches to simplifying such signatures.

Avoiding substructures. Strictly speaking, a signature need never specify substructures, even if it must be self-contained. Instead, it can specify all the types appearing in its `val` specifications. Omitting structure *Braun* from our signature makes it more readable:

```
signature BRAUNPAIR1 =
  sig
  type 'a tree
  val zip: 'a tree * 'b tree -> ('a*'b) tree
  ⋮
  end;
```

This signature specifies considerably less than *BRAUNPAIR0*. All the components of *Braun* are missing, and *tree* is specified as a mere `type`. Specifying *tree* as a `datatype` would have required copying its entire specification from signature *TREE*, an unpleasant duplication.

A signature should be as small as possible, so *BRAUNPAIR1* may be ideal. It is, provided the components it specifies can be used independently of those in *Braun*; in other words, it should be self-contained in use, rather than in the formal sense of having no free identifiers.

Sharing constraints in a signature. A signature should be as small as possible, but it should not be even smaller. If every instance of *BRAUNPAIR1* must be accompanied by an instance of *BRAUN*, then identifying their *tree* components will require a sharing constraint. Every functor heading of the form

```
functor PairFunctor0 (BP: BRAUNPAIR0)
```

could become more than twice as long:

```
functor PairFunctor1 (structure Braun: BRAUN
                      structure BP: BRAUNPAIR1
                      sharing type Braun.Tree.tree = BP.tree)
```

The solution is to specify both substructure *Braun* and type *tree* in the same signature, with a sharing constraint to relate them:

```
signature BRAUNPAIR2 =
  sig
  structure Braun: BRAUN
  type 'a tree
  sharing type tree = Braun.Tree.tree
  val zip: 'a tree * 'b tree -> ('a*'b) tree
  ⋮
  end;
```

A structure that matches this signature must declare type α *tree* to satisfy the sharing constraint:

```
type 'a tree = 'a Braun.Tree.tree
```

We have the advantages of both the previous approaches. The signature is readable and complete, allowing simple functor headings like that of *PairFunctor*0.

Type abbreviations in signatures. Type sharing constraints are adequate for the present purpose, namely to shorten compound names in signatures. But they cannot specify arbitrary type abbreviations. Sharing constraints apply to identifiers, not to types; the sharing specification is

```
tree = Braun.Tree.tree
```

and not

```
'a tree = 'a Braun.Tree.tree
```

Signatures may make type abbreviations. This is yet another way of shortening those compound names:

```
signature BRAUNPAIR3 =
    sig
    structure Braun: BRAUN
    type 'a tree = 'a Braun.Tree.tree
    val zip: 'a tree * 'b tree -> ('a*'b) tree
    end;
```

For a structure to match this signature, it must declare an equivalent type abbreviation.

The `include` *specification.* ***Including*** a signature means specifying its components as belonging directly to the current signature, not to a substructure. The specification

```
include SIG
```

has the effect of writing out the contents of *SIG* without the surrounding `sig` ...`end` brackets. Our example now becomes

```
signature BRAUNPAIR4 =
  sig
  include BRAUN
  val zip: 'a Tree.tree * 'b Tree.tree -> ('a*'b) Tree.tree
  end;
```

There are compound names, but they are acceptably short because substructure *Braun* has vanished. All its components have been incorporated into the new signature. So instances of *BRAUNPAIR4* match signature *BRAUN* as well.

Including yourself in trouble. Multiple inclusion can be a powerful structuring technique. It can be combined with sharing constraints to get a limited effect of renaming. If the included signature specifies, say, types *to* and *from*, then sharing constraints can identify these types with other types in the signature (Biagioni *et al.*, 1994). The standard library uses sharing constraints in a similar fashion, sometimes to rename the components of substructures.

Avoid including signatures that have names in common: this could give an identifier repeated or conflicting specifications. Excessive use of `include` can lead to large, flat signatures, obscuring the module hierarchy. If signature *BRAUN* had itself specified

```
include TREE
```

instead of

```
structure Tree: TREE
```

then we should have no compound names at all. In a superficial sense, this would aid readability. But all the components of three different structures would be thrown together without any organization.

Reference guide to modules

This section collects the concepts of structures, signatures and functors, summarizing the modules language as a whole. It describes practically the entire language, including some of the more obscure features. First, let us review the basic definitions.

A ***structure*** is a collection of declarations, typically of items that serve a common purpose. These may include types, values and other structures. Since structures may be grouped into larger structures, a software system can be designed as a hierarchy. A structure can be treated as a unit, no matter how complex it is internally.

A ***signature*** consists of type checking information about each item declared in a structure. It lists the types; it lists the values, with their types; it lists the substructures, with their signatures. Sharing constraints identify common components of substructures. Just as different values can have the same type, different structures can have the same signature.

A ***functor*** is a mapping from structures to structures. The body of the functor defines a structure in terms of a formal parameter, which is specified by a signature. Applying the functor substitutes an actual structure into the body. Functors allow program units to be coded separately and can express generic units.

A ***module*** is either a structure or a functor.

7.16 *The syntax of signatures and structures*

This book aims to teach programming techniques, not to describe Standard ML in full. Modules involve a great deal of syntax, however; here is a systematic description of their main features.

In the syntax definitions, an optional phrase is enclosed in square brackets. A repeatable phrase (occurring at least once) is indicated informally using three dots (...). For example, in

$$\texttt{exception}\ Id_1\ \left[\texttt{of}\ T_1\right]\ \texttt{and}\ \ldots\ \texttt{and}\ Id_n\ \left[\texttt{of}\ T_n\right]$$

the '`of` T_1' phrases are optional. The keyword `and` separates simultaneous declarations.

Syntax of signatures. A signature has the form

`sig` *Spec* `end`

where a *Spec* is a specification of types, values, exceptions, structures and sharing constraints.

A value specification of the form

`val` $Id_1 : T_1$ `and ... and` $Id_n : T_n$

specifies values named $Id_1, \ldots, Id_n$ with types $T_1, \ldots, T_n$, respectively. Several values and their types can be specified simultaneously.

Types may be specified (simultaneously) by

`type` $\left[TypeVars_1 \right] Id_1 \left[= T_1 \right]$ `and ... and` $\left[TypeVars_n \right] Id_n \left[= T_n \right]$

If T_i is present, for $i = 1, \ldots, n$, then Id_i is specified as a type abbreviation.

Types that admit equality may be specified by

`eqtype` $\left[TypeVars_1 \right] Id_1$ `and ... and` $\left[TypeVars_n \right] Id_n$

In both `type` and `eqtype` specifications, a type is given by optional type variables ($TypeVars$) followed by an identifier, exactly as may appear on the left side of a type declaration. A `datatype` specification has the same syntax as a `datatype` declaration.

Exceptions, with optional types, can be specified by

`exception` $Id_1 \left[\texttt{of } T_1 \right]$ `and ... and` $Id_n \left[\texttt{of } T_n \right]$

Structures, with their signatures, are specified by

`structure` $Id_1 : Sig_1$ `and ... and` $Id_n : Sig_n$

Sharing constraints have the form

`sharing` $\left[\texttt{type} \right] Id_1 = Id_2 = \cdots = Id_n$

The identifiers $Id_1, \ldots, Id_n$ are specified to share. If the keyword `type` is present then they must be type identifiers; otherwise they must be structure identifiers. Sharing constraints may appear in any signature; they most frequently appear in a functor's formal parameter list, which is given as a signature specification. Sharing of two structures implies sharing of their corresponding named types.

Include specifications have the form

`include` $SigId_1 \cdots SigId_n$

Each $SigId$ should be a signature identifier, and specifies the components of that signature.

The `where type` qualification. A new signature form has recently been proposed, which allows us to constrain type identifiers Id_i to existing types T_i in a signature Sig:

Sig `where type` $\left[TypeVars_1\right]Id_1$ `=` T_1 `and` $\left[TypeVars_n\right]Id_n$ `=` T_n

This construct can be used wherever signatures are combined in elaborate ways. In conjunction with an opaque signature constraint, it provides another way of declaring abstract types. Consider this functor heading:

```
functor Dictionary (Key: ORDER)
        :> DICTIONARY where type key = Key.t
```

The functor's result signature is an abstract view of *DICTIONARY*, but with key constrained to be the type of sort keys specified by the argument structure Key. This corrects the 'all-or-nothing' limitation of opaque constraints mentioned at the end of Section 7.5. The functor body no longer has to use `abstype`.

Syntax of structures. A structure can be created by a declaration (which may declare substructures) enclosed by the brackets `struct` and `end`:

`struct` D `end`

A structure can also be given by a functor application:

$FunctorId$ `(`Str`)`

The functor named $FunctorId$ is applied to the structure Str. This is the primitive syntax for functor application, used in our first examples, which allows only one argument. Passing multiple arguments requires the general form of functor application, where the argument is a declaration:

$FunctorId$ `(`D`)`

This abbreviates the functor application

$FunctorId$ `(struct` D `end)`

and is analogous to writing a tuple as the argument of a function for the effect of multiple arguments.

Local declarations in a structure have the form

`let` D `in` Str `end`

Evaluation performs the declaration D and yields the value of the structure expression Str. The scope of D is restricted to Str.

A structure may have a transparent or opaque signature constraint:

Str `:` Sig
Str `:>` Sig

7.17 *The syntax of module declarations*

Signature, structure and functor declarations are not allowed within expressions. Structures may be declared inside other structures, but functor declarations must not be nested.

Signature constraints are given in the form `:>`Sig but `:`Sig is also allowed.

A signature declaration makes the identifiers Id_1, ..., Id_n denote the signatures $Sig_1, \ldots, Sig_n$, respectively:

$$\texttt{signature}\ Id_1\texttt{=}Sig_1\ \texttt{and}\ \ldots\ \texttt{and}\ Id_n\texttt{=}Sig_n$$

A structure declaration makes the identifier Id_i denote the structure Str_i (optionally specifying the signature Sig_i), for $1 \leq i \leq n$:

$$\texttt{structure}\ Id_1\Big[\texttt{:>}Sig_1\Big]\texttt{=}Str_1\ \texttt{and}\ \ldots\ \texttt{and}\ Id_n\Big[\texttt{:>}Sig_n\Big]\texttt{=}Str_n$$

The primitive syntax for a functor declaration is

$$\texttt{functor}\ Id\ \ \texttt{(}Id'\texttt{:}Sig'\texttt{)}\ \ \Big[\texttt{:>}Sig\Big]\ \ \texttt{=}\ \ Str$$

where *Id* is the name of the functor, *Id′* and *Sig′* are the name and signature of the formal parameter, structure *Str* is the body and *Sig* is an optional signature constraint.

The general syntax for a functor declaration, which gives the effect of multiple arguments, has the form

$$\texttt{functor}\ Id\ \ \texttt{(}Spec\texttt{)}\ \ \Big[\texttt{:>}Sig\Big]\ \ \texttt{=}\ \ Str$$

The formal parameter list is given by the specification *Spec*. The functor still takes one argument, a structure whose signature is determined by *Spec*. The formal parameter is implicitly opened in the body of the functor, making its components visible.

Summary of main points

- Structures do not hide internal representations.
- The `abstype` declaration can be combined with structures and signatures to hide the internal details of an abstract data type.
- A functor is a structure that takes other structures as parameters.
- Functors can express generic algorithms and permit program units to be combined freely.
- Sharing constraints may be necessary to ensure that certain subcomponents of a system are identical.

- Compound names can be abbreviated by careful use of `open` declarations, among other methods.