Sample Code for this Chapter

It is good practice to ascribe a signature to every structure binding in a program to ensure that the signature of the bound structure variable is apparent from the binding. In the preceding chapter we described the elaboration and evaluation of a structure binding with an explicit signature ascription. First the ascribed signature is used to determine a *view* of the principal signature of the right-hand side of the binding, then the view is checked to ensure that it verifies the type sharing requirements of the ascribed signature. If both steps succeed, we assign a signature to the bound structure variable according to whether it is a transparent or opaque ascription --- if it is transparent, we assign the view to the variable, otherwise the ascription. Thus transparent ascription is used to form views of a structure, and opaque ascription is used to form abstractions in which critical type information is hidden from the rest of the program.

The formation of a view also has significance at run-time: a new structure is built consisting of only those components of the right-hand side of the binding mentioned in the ascribed signature, perhaps augmented by zero or more type components to ensure that the signature of the view is well-formed. (For example, if we attempt to extract only the constructors of a datatype, and not the datatype itself, the compiler will implicitly extract the datatype to ensure that the types of the constructors are expressible in the signature. Any type implicitly included in the view is marked as "hidden" to indicate that it was implicitly included as a consequence of the explicit inclusion of some other components of the structure.) Moreover, the types of polymorphic value components may be specialized in the view, corresponding to a form of polymorphic instantiation during signature matching. The result is a structure whose shape is fully determined by the view; no "junk" remains after the ascription. This ensures that access to the components of a structure is efficient (constant-time), and that there are no "space leaks" stemming from the presence of components of a structure that are not mentioned in its signature.

In this chapter we discuss the trade-off's between using views and abstraction in ML by offering some guidelines and examples of their use in practice. How does one decide whether to use transparent or opaque ascription? Generally speaking, transparent ascription is appropriate if the signature is not intended to be exhaustive, but is rather just a specification of some minimum requirements that a module must satisfy. Opaque ascription is appropriate if the signature is intended to be exhaustive, specifying precisely the operations that are available on the type.

Here's a common example of the use of transparent ascription in a program. When defining a module it is often convenient to introduce a number of auxiliary bindings, especially of "helper functions" that are used internally to the code of the "public" operations. Since these auxiliaries are not intended to be used by clients of the module, it is good practice to localize them to the implementation of the public operations. This can be achieved by using the `local` construct, as previously discussed in these notes. An alternative is to define the auxiliaries as components of the stucture, relying on transparent ascription to drop the auxiliary components before exporting the public components to clients of the module. Thus we might write something like this:

```
structure IntListOrd : ORDERED =
struct
  type t = int list
  fun aux l = ...
  val lt (l1, l2) = ... aux ...
  val eq (l1, l2) = ... aux ...
end
```

The effect of the signature ascription is to drop the auxiliary component `aux` from the structure during signature matching so that afterwards the binding of `IntListOrd` contains only the components in the signature `ORDERED`. An added bonus of this style of programming is that during debugging and testing we may gain access to the auxiliary by simply "commenting out" the ascription by writing instead

```
structure IntListOrd (* : ORDERED *) =
struct
  type t = int list
  fun aux l = ...
  val lt (l1, l2) = ... aux ...
  val eq (l1, l2) = ... aux ...
end
```

Since the ascription has been suppressed, the auxiliary component `IntListOrd.aux` is accessible for testing. (It would be useful to have a compiler switch that "turns off" signature ascription, rather than having to manually comment out each ascription in the program, but no current compilers support such a feature.)

Now let us consider uses of opaque ascription by reconsidering the implementation of persistent queues using pairs of lists. Here it makes sense to use opaque ascription since the operations specified in the signature are intended to be exhaustive --- the only way to create and manipulate queues is to use the operations `empty`, `insert`, and `remove`. By using opaque signature matching in the declaration of the `Queue`

structure, we ensure that the type `Queue.queue` is hidden from the client. Consequently an expression such as `Queue.insert (1, ([],[]))` is ill-typed, even though queues are "really" pairs of lists, because the type `'a list * 'a list` is not equivalent to `'a Queue.queue`. Were we to use transparent ascription this equation would hold, which means that the client would not be constrained to using only the "official" queue operations on values of type `'a Queue.queue`. This violates the principle of *data abstraction*, which states that an abstract type should be completely defined by the operations that may be performed on it.

Why impose such a restriction? One reason is that it ensures that the client of an abstraction is insensitive to changes in the implementation of the abstraction. Should the client's behavior change as a result of a change of implementation of an abstract type, we know right where to look for the error: it can *only* be because of an error in the implementation of the operations of the type. Were abstraction not enforced, the client might (accidentally or deliberately) rely on the implementation details of the abstraction, and would therefore need to be modified whenever the implementation of the abstraction changes. Whenever such coupling can be avoided, it is desirable to do so, since it allows components of a program to be managed independently of one another.

A closely related reason to employ data abstraction is that it enables us to enforce representation invariants on a data structure. More precisely, it enables us to isolate any violations of a representation invariant to the implementation of the abstraction itself. No client code can disrupt the invariant if abstraction is enforced. For example, suppose that we are implementing a dictionary package using a binary search tree. The implementation might be defined in terms of a library of operations for manipulating generic binary trees called `BinTree`. The implementation of the dictionary might look like this:

```
structure Dict :> STRING_DICT =
  struct
    (* Rep Invariant: binary search tree *)
    type t = string BinTree.tree
    fun insert (k, t) = ...
    fun lookup k = ...
  end
```

Had we used transparent, rather than opaque, ascription of the `STRING_DICT` signature to the `Dict` structure, the type `Dict.t` would be known to clients to be string `BinTree.tree`. But then one could call `Dict.lookup` with any value of type `string BinTree.tree`, not just one that satisfies the representation invariant governing binary search trees (namely, that the strings at the nodes descending from the left child of a node are smaller than those at the node, and those at nodes descending from the right child are larger than those at the node). By using opaque ascription we are isolating the implementation type to the `Dict` package, which means that the only possible violations of the representation invariant are those that arise from errors in the `Dict` package itself; the invariant cannot be disrupted by any other means. The operations themselves may *assume* that the representation invariant holds whenever the function is called, and are obliged to *ensure* that the representation invariant holds whenever a value of the representation type is returned. Therefore any combination of calls to these operations yielding a value of type `Dict.t` must satisfy the invariant.

You might wonder whether we could equally well use run-time checks to enforce representation invariants. The idea would be to introduce a "debug flag" that, when set, causes the operations of the dictionary to check that the representation invariant holds of their arguments and results. In the case of a binary search tree this is surely possible, but at considerable expense since the time required to check the binary search tree invariant is proportional to the size of the binary search tree itself, whereas an insert (for example) can be performed in logarithmic time. But wouldn't we turn off the debug flag before shipping the production copy of the code? Yes, indeed, but then the benefits of checking are lost for the code we care about most! (To paraphrase Tony Hoare, it's as if we used our life jackets while learning to sail on a pond, then tossed them away when we set out to sea.) By using the type system to enforce abstraction, we can confine the possible violations of the representation invariant to the dictionary package itself, and, moreover, we need not turn off the check for production code because there is no run-time penalty for doing so.

A more subtle point is that it may not always be possible to enforce data abstraction at run-time. Efficiency considerations aside, you might think that we can always replace static localization of representation errors by dynamic checks for violations of them. But this is false! One reason is that the representation invariant might not be computable. As an example, consider an abstract type of *total* functions on the integers, those that are guaranteed to terminate when called, without performing any I/O or having any other computational effect. It is a theorem of recursion theory that no run-time check can be defined that ensures that a given integer-valued function is total. Yet we can define an abstract type of total functions that, while not admitting ever possible total function on the integers as values, provides a useful set of such functions as elements of a structure. By using these specified operations to create a total function, we are in effect encoding a proof of totality in the code itself.

Here's a sketch of such a package:

```
signature TIF = sig
  type tif
  val apply : tif -> (int -> int)
  val id : tif
  val compose : tif * tif -> tif
  val double : tif
  ...
end

structure Tif :> TIF = struct
  type tif = int->int
  fun apply t n = t n
  fun id x = x
  fun compose (f, g) = f o g
  fun double x = 2 * x
  ...
end
```

Should the application of such some value of type `Tif.tif` fail to terminate, we know where to look for the error. No run-time check can assure us that an arbitrary integer function is in fact total.

Another reason why a run-time checkto enforce data abstraction is impossible is that it may not be possible to tell from looking at a given value whether or not it is a legitimate value of the abstact type. Here's an example. In many operating systems processes are "named" by integer-value process identifiers. Using the process identifier we may send messages to the process, cause it to terminate, or perform any number of other operations on it. The thing to notice here is that any integer at all is a possible process identifier; we cannot tell by looking at the integer whether it is indeed valid. No run-time check on the value will reveal whether a given integer is a "real" or "bogus" process identifier. The only way to know is to consider the "history" of how that integer came into being, and what operations were performed on it. Using the abstraction mechanisms just described, we can enforce the requirement that a value of type `pid`, whose underlying representation is `int`, is indeed a process identifier. You are invited to imagine how this might be achieved in ML.

Transparency and opacity may seem, at first glance, to be fundamentally opposed to one another. But in fact transparency is *special case* of opacity! By using type definitions in signatures, we may always express *explicitly* the propagation of type information that is conveyed *implicitly* by transparent ascription. For example, rather than write

```
structure IntLT : ORDERED = struct type t=int ... end
```

we may instead write

```
structure IntLT :> INT_ORDERED = struct type t=int ... end
```

at the expense of introducing a specialized version of the signature `ORDERED` with the type `t` defined to be `int`. This syntactic inconvenience can be ameliorated by using the "`where type`" construct, writing

```
structure IntLT :> ORDERED where type t=int = struct ... end
```

The signature expression "`ORDERED where type t=int`" is equivalent to the signature `INT_ORDERED` defined above.

Thus transparency is a form of opacity in which we happen to publicize the identity of the underlying types in the ascribed signature. This observation is more important than one might think at first glance. The reason is that it is often the case that we must use a combination of opacity and transparency in a given situation. Here's an example. Suppose that we wished to implement several dictionary packages that differ in the type of keys. The "generic" signature of a dictionary might look like this:

```
signature DICT = sig
  type key
  val lt : key * key -> bool
  val eq : key * key -> bool
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

Notice that we include a type component for the keys, together with operations for comparing them, along with the type of dictionaries itself and the operations on it. Now consider the definition of an integer dictionary module, one whose keys are integers ordered in the usual manner. We might use a declaration like this:

```
structure IntDict :> DICT = struct
  type key = int
  val lt : key * key -> bool = (op <)
  val eq : key * key -> bool = (op =)
  datatype 'a dict = Empty | Node of 'a dict * 'a * 'a dict
```

```
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

But this doesn't work as intended! The reason is that the opaque ascription, which is intended to hide the implementation type of the abstraction, also obscures the type of keys. Since the only operations on keys in the signature are the comparison functions, we can never insert an element into the dictionary!

What is necessary is to introduce a specialized version of the `DICT` signature in which we publicize the identity of the `key` type, as follows:

```
signature INT_DICT = DICT where type key = int

structure IntDict :> INT_DICT = struct
  type key = int
  val lt : key * key -> bool = (op <)
  val eq : key * key -> bool = (op =)
  datatype 'a dict = Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

With this declaration the type `'a IntDict.dict` is abstract, but the type `IntDict.key` is equivalent to `int`. Thus we may correctly write `IntDict.insert (IntDict.empty, 1, "1")` to insert the value `"1"` into the empty dictionary with key `1`. To build a dictionary whose keys are strings, we proceed similarly:

```
signature STRING_DICT = DICT where type key = string

structure StringDict :> STRING_DICT = struct
  type key = string
  val lt : key * key -> bool = (op <)
  val eq : key * key -> bool = (op =)
  datatype 'a dict = Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

In the next two chapters we will discuss how to build a *generic* implementation of dictionaries that may be instantiated for many different choices of key type.

Sample Code for this Chapter