Chapter 1

The Design Space of ML Modules

What is the ML module system? It is difficult to say. There are several dialects of the ML language, and while the module systems of these dialects are certainly far more alike than not, there are important and rather subtle differences among them, particularly with regard to the semantics of data abstraction. The goal of Part I of this thesis is to offer a new way of understanding these differences, and to derive from that understanding a unifying module system that harmonizes and improves on the existing designs.

In this chapter, I will give an overview of the existing ML module system design space. I begin in Section 1.1 by developing a simple example—a module implementing sets—that establishes some basic terminology and illustrates some of the key features shared by all the modern variants of the ML module system. Then, in Section 1.2, I describe several dialects that represent key points in the design space, and discuss the major axes along which they differ.

1.1 Key Features of the ML Module System

1.1.1 Structures and Signatures

In ML, code and data are grouped together in *modules*. The basic module construct is called a *structure*, and Figure 1.1 shows an example of a structure implementing integer sets.[1] The structure `IntSet` is defined by a structure expression `struct ... end`, which contains a sequence of bindings. The first binding defines the type name `set` as an abbreviation for the type `int list` of integer lists, thus indicating that sets are being implemented by this module as lists. Type bindings are much like typedefs in C; the type `set` and the type `int list` are interchangeable. The second binding in `IntSet` is a value binding, defining `emptyset` to be the empty list `[]`, which has type `set` because it has type `int list`. The remaining bindings are function bindings: an `insert` operation that takes an integer and a set and returns the result of pushing the integer onto the front of the list representing the set, and a `member` operation that checks whether an integer belongs to a set by performing a sequential search on the list representing the set.[2] Although this example does not illustrate it, structures in ML may also contain substructure bindings, thereby allowing modules to be built up as composites of other modules and enabling flexible namespace management.

Now that we have defined this module `IntSet`, we can use it essentially as we would use an object in Java or a `struct` in C—by projecting out its components using the "dot notation." For

[1] This example, as well as the others in this section, is written in Standard ML syntax.

[2] Note that, in keeping with functional programming style, this is a persistent implementation of sets, *e.g.,* inserting an integer into a set does not modify the input set but merely returns a new set containing the integer.

```
structure IntSet =
  struct
    type set = int list
    val emptyset : set = []
    fun insert (x : int, S : set) : set = x::S
    fun member (x : int, S : set) : bool = ...
    ...
  end
```

Figure 1.1: ML Module for Integer Sets

```
signature INT_SET =
  sig
    type set
    val emptyset : set
    val insert : int * set -> set
    val member : int * set -> bool
    ...
  end
```

Figure 1.2: ML Interface for Integer Sets

instance, we might define the set `S` by the following value binding:

```
val S : IntSet.set = IntSet.insert(5, IntSet.emptyset)
```

This defines `S` by inserting 5 into the empty set. A distinguishing feature of ML modules is that, in addition to having data and function components, they have type components, such as the `set` type component of `IntSet`. Correspondingly, we can also use the dot notation to project out the type `IntSet.set`, which is the return type of `IntSet.insert` and thus the type of `S`.

As mentioned above, `IntSet.set` is merely an abbreviation for `int list`. However, there is no need for clients of the `IntSet` module to know this. In the interest of data abstraction, we would thus like to hide the knowledge that `IntSet.set` is equivalent to `int list`. This is achieved by first defining an *interface* that describes what the clients *do* need to know. In ML, interfaces are called *signatures*, and Figure 1.2 shows an appropriately abstract signature for integer sets.

The signature `INT_SET` is defined by a signature expression `sig` ... `end`, which contains a list of specifications for the components of the `IntSet` module. The specifications for `emptyset`, `insert` and `member` are straightforward, assigning to each value component a type. (In a functional language like ML, function components are just value components with arrow types.) The interesting specification is the one for the `set` type, which holds its definition abstract. A more precise interface for the `IntSet` module would replace `INT_SET`'s abstract specification of the `set` component with the transparent specification `type set = int list`, which exposes the implementation of sets as lists. ML allows one to specify type components in interfaces with or without their definitions, thus providing fine-grained control over the propagation of type information (see Section 1.1.3). In this case, however, the abstract `INT_SET` interface is a more appropriate description of sets, as it does not allow clients to depend on any particular implementation strategy.

```
signature COMPARABLE =
  sig
    type item
    val compare : item * item -> order
  end

functor Set (Item : COMPARABLE) =
  struct
    type set = Item.item list
    val emptyset : set = []
    fun insert (x : int, S : set) : set = x::S
    fun member (x : int, S : set) : bool =
      ... Item.compare(x,y) ...
    ...
  end
```

Figure 1.3: ML Functor for Generic Sets

1.1.2 Data Abstraction via Sealing and Functors

Just defining the `INT_SET` signature does not do anything by itself. To ensure that the clients' view of `IntSet` is limited to what appears in `INT_SET`, we must *seal* `IntSet` with `INT_SET`, as follows:

```
structure IntSet = IntSet :> INT_SET
```

Given this new definition for `IntSet`, we may still project out the type `IntSet.set`, but it is not known to be equivalent to `int list`. Consequently, the only way clients of `IntSet` can create values of type `IntSet.set` is by using `insert` and `emptyset` (and presumably other operations like `union` and `intersection`), which are explicitly specified in `INT_SET`. Although the `IntSet` example does not illustrate it, sealing can also be used to hide the existence of certain value components in a module. We will see an example of this in Section 1.2.6.

Another distinguishing feature of the ML module system is its *functor* mechanism. Functors are simply functions from modules to modules. Much as functions allow a piece of code to be reused with different instantiations of its parameters, functors allow a module to be reused with different instantiations of the modules it depends on. To continue the `IntSet` example, the implementation of sets is largely indifferent to the type of items stored in the sets and would be more useful if it were not restricted to sets of integers. The only reason the type of items matters at all is that the implementation of a function like `member` assumes an ordering on items. Functors allow us to make the implementation of sets generic with respect to the item type, as shown in Figure 1.3.

First, we define a signature `COMPARABLE` describing what the set module needs to know about the item type. This signature characterizes modules that provide some type `item`—which one it is is irrelevant—together with a function `compare` for ordering values of that type. The `Set` module is then defined as a functor that takes as input a module `Item` of signature `COMPARABLE` and returns a module implementing sets containing items of type `Item.item`. Note that the `set` type is now defined as `Item.item list`, and the `member` function invokes `Item.compare` for ordering `Item.item`'s instead of relying on integer comparison.

We can now generate sets of different item types very easily. For example, as shown in Figure 1.4, we can reproduce the functionality of our original `IntSet` module by first defining a module

```
structure IntItem =
  struct
    type item = int
    fun compare (x,y) = Int.compare(x,y)
  end
structure StringItem =
  struct
    type item = string
    fun compare (x,y) = String.compare(x,y)
  end

structure IntSet = Set(IntItem)
structure StringSet = Set(StringItem)
```

Figure 1.4: Instantiating the Set Functor

`IntItem`, which provides `int` as the `item` type along with the built-in integer comparison function, and then applying `Set` to `IntItem`. If we want to generate an implementation of integer sets based on a different ordering of the integers, we can apply the `Set` functor to an item module with the same definition of the `item` type but with a different comparison function. A module implementing sets of strings or any other type can also be generated in a similar manner.

To summarize, we have seen two mechanisms that ML provides for supporting data abstraction. First, the sealing mechanism supports *implementor-side* data abstraction by allowing the implementor of the set module to hide information about the implementation of sets from its clients. Second, by thinking of the set module as itself being a client of an item module with an abstract interface, we see that ML functors exploit the idea of *client-side* data abstraction to provide a powerful form of code reuse.

1.1.3 Translucent Signatures

The natural next step in the development of the `Set` example is to combine ML's two forms of data abstraction by sealing the body of the `Set` functor with an abstract signature that hides the implementation of sets as lists. The question is what signature to use. Now that we have generalized the implementation of sets to support an arbitrary item type, the `INT_SET` signature is no longer applicable. The most obvious answer is to use a signature that replaces all occurrences of `int` in `INT_SET` with references to `Item.item`. However, the type `Item.item` only makes sense inside the body of the `Set` functor, and we would like to be able to define a generic signature for sets separately from this particular implementation of them.

One way to define a generic interface for sets would be to allow a signature to be parameterized by a module [37], in which case one could define a parameterized signature `SET` as follows:

```
signature SET (Item : COMPARABLE) =
  sig
    type set
    ...
    val insert : Item.item * set -> set
    ...
  end
```

```
signature SET =
  sig
    type item
    type set
    val emptyset : set
    val insert : item * set -> set
    val member : item * set -> bool
    ...
  end
```

Figure 1.5: Generic ML Signature for Sets

```
functor Set (Item : COMPARABLE) =
  struct
    type item = Item.item
    type set = item list
    ... (* same as before *) ...
  end
  :> SET where type item = Item.item
```

Figure 1.6: Sealed ML Functor for Generic Sets

The body of the `Set` functor could then be sealed with the signature `SET(Item)`.

In fact, however, one need not introduce an explicit form of parameterized signature in order to characterize a generic interface for sets. ML provides implicit support for parameterized signatures through the idea of *translucency*. As we have seen, type components in ML signatures may be specified "opaquely" (*e.g.,* `type set`), but they may also be specified "transparently" (*e.g.,* `type set = int list`). Signatures that support both kinds of specifications are known as "translucent."

Figure 1.5 shows the signature for generic sets that one would write in the ML style. Instead of making the `item` type a parameter of the `SET` signature, the ML approach is to include `item` as an abstract type component in the signature. In other words, a module implementing sets carries the type of items along with it, whatever that type may be, thus enabling the generic interface for sets to be self-contained.

Figure 1.6 shows how the implementation of the `Set` functor is made abstract. The `item` component of the `Item` argument is copied into the body of the functor; the body is then sealed with the signature `SET where type item = Item.item`, which is shorthand in ML for the signature formed by taking the abstract `SET` signature and making the `item` component transparently equal to `Item.item`. Thus, the signature with which the body of the `Set` functor is sealed is translucent—it reveals the identity of the `item` type, which is necessary in order for the resulting set module to be of any use, but it holds the identity of the `set` type abstract.

Translucency subsumes the utility of parameterized signatures, but it is useful for other reasons as well. First, it allows one to reveal partial information about the identity of a type. For instance, suppose a module exports a type `t` which is defined internally to be `int * string`, and the implementor of the module wishes to reveal that values of type `t` are pairs whose first component is an integer, but does not wish to reveal that the second component is a string. Then the implementor

can seal the module with a signature containing an opaque `type u`, which is defined internally to be `string`, and a transparent `type t = int * u`.

In addition, the support for transparent type specifications in signatures means that for most modules in ML there is a *principal signature*, *i.e.,* a most-specific signature that encapsulates all that can be observed about the module during typechecking.[3] For example, the principal signature of the module `IntItem` defined in Figure 1.4 is `COMPARABLE where type item = int`. The existence of principal signatures is advantageous for modular program development because it allows a program to be divided at relatively arbitrary points, with the assurance that all the typing information about any one component of the program is expressible in the form of a signature that the programmer can write independently of the implementation of that component.

Lastly, translucency accounts naturally for the concept of *type sharing*. It often happens that one wants to take as input to a functor two modules (call them `A` and `B`), each of which provides a type component `t`, and in order for the body of the functor to make any sense it is necessary that `A.t` is equal to `B.t`. ML supports such a "type sharing" constraint by letting the programmer attach `sharing type A.t = B.t` to the specification of the functor arguments. In earlier versions of ML, such as SML '90, type sharing constraints provided an increase in expressive power that proved difficult to account for in type-theoretic studies of the module system [47, 29]. In modern dialects of ML, however, type sharing can be seen as just an instance of translucency. The constraint `sharing type A.t = B.t` can be seen as syntactic sugar that has the effect of modifying the signature of argument `B` so that its type component `t` is specified transparently as `type t = A.t`.[4]

For further illustrations of the power of translucent signatures, I refer the reader to one of the more pedagogical treatments of ML programming that are available, such as Harper [26].

1.2 Key Points and Axes in the Design Space of ML Modules

Since its inception [46], the ML module system has been associated with the mechanisms of *signatures*, *structures* and *functors*. The *sealing* mechanism[5] was proposed early on by MacQueen in the form of an `abstraction` binding, which was implemented in 1993 in an early version of the SML/NJ compiler (version 0.93) [71]. Translucent signatures were also implemented in version 0.93 of SML/NJ, but were not treated formally until 1994, when Harper and Lillibridge [28] and Leroy [42] independently proposed similar formalisms for them at the same POPL symposium.

Although the formal accounts prior to that point still provide much valuable insight—particularly Harper, Mitchell and Moggi's work on higher-order modules [30], which introduces the concept of *phase separation* that underlies much of the analysis in Chapter 2 and the formal system in Chapter 4—I am focused in this thesis on accounting for the semantic variations among the "modern" variants of the ML module system that support all of the features described in Section 1.1, including translucency. In this section I will describe several such variants and how they relate to one another in the design space of ML modules. I will begin, though, with a bit of historical context.

[3] As we will see in Section 4.2.6, due to the "avoidance problem," not all modules in ML have principal signatures, but there is a considerable subset of ML in which modules do have principal signatures. Also, to avoid confusion, it is worth noting that the Commentary on the original Definition of SML [50] also uses the term *principal signature*, but to describe a concept unrelated to the notion of fully-descriptive signature intended here.

[4] Or, if module `B` comes before `A` in the order of the functor arguments, modifying the signature of `A` so that its type component `t` is specified transparently as `type t = B.t`. For details on how type sharing constraints may be desugared in general, see Chapter 9.

[5] I mean *sealing* here in its "opaque" form (`:>`), as described in Section 1.1.2. In contrast, the "transparent" form of sealing (`:`) was part of the Definition from the beginning, but does not provide full support for data abstraction and is not present in other dialects of ML, such as O'Caml. For further discussion of the difference between opaque and transparent sealing, see Section 9.3.2.

1.2.1 Precursors to Translucency

The idea of translucency present in modern variants of ML arose in response to a bifurcation that had developed in the late '80s and early '90s in the semantics of modularity. On one hand there was the approach taken by SML '90 [51], which is modeled in formal accounts by MacQueen [47] and Harper *et al.* [29, 30] in terms of "strong sum" types. While it supports client-side data abstraction via functors, SML '90 does not fully support implementor-side data abstraction. In particular, sealing in SML '90 is "transparent," that is, sealing a module with a signature limits which components of the module are externally visible but does not hide the definitions of any visible type components, even those that are specified opaquely in the signature. (Lillibridge correspondingly termed the SML '90 approach the "transparent" approach to modularity [45].) In addition, as I have already mentioned, the type-theoretic treatments of SML '90-style modularity were not able to account for the idea of type sharing constraints in signatures.

On the other hand there was the "opaque" approach, due to Mitchell and Plotkin [53], in which abstract data types are modeled as existential types. Existentials provide an elegant logical foundation for type abstraction and, unlike the transparent approach, provide full support for implementor-side data abstraction. They are awkward, however, as a basis for modular program construction. In particular, a value of existential type is not as flexible as an ML module. One cannot refer to the abstract types and associated operations provided by such a value via the standard "dot notation" (*e.g.,* `IntSet.insert`) used for modules. Rather, in order to use a value v of type $\exists\alpha.\mathrm{C}$, one must "open" or "unpack" v—as in the expression "`open` v `as` $[\alpha, x]$ `in` e"—in which case the scope of the abstract type α and of the associated operations represented by x (of type C) is limited to the expression e. In contrast, unless otherwise delimited, the scope of the types and values provided by an ML module is "the rest of the program," which may not even have been written yet. Furthermore, whereas the transparent approach suffers from allowing too much type information to be propagated, the opaque approach suffers from not allowing enough. For example, if one applies the identity function id to a value v of type $\exists\alpha.\mathrm{C}$, there is no way to tell that v and $id(v)$ share their abstract type component because there is no way to even refer to v's abstract type component. In contrast, applying the identity functor to the `IntSet` module in ML (even in SML '90) will result in a module whose `set` type is transparently equal to `IntSet.set`.

As illustrated in Section 1.1.3, translucent signatures and opaque sealing address the deficiencies of both the opaque and the transparent approaches to modularity, combining the flexibility of ML-style modules with the support for implementor-side abstraction provided by existentials.[6] Although Harper and Lillibridge [28] and Leroy [42] differ in their terminology, the former speaking of "translucent sums" and the latter of "manifest types," the basic idea of translucency put forth by both papers is the same. The key point that distinguishes these two accounts is that Harper and Lillibridge's supports *first-class* modules whereas Leroy's supports *second-class* modules.

1.2.2 First-Class vs. Second-Class, Higher-Order vs. First-Order

The primary feature that distinguishes functional programming languages from other kinds of languages is that in functional languages functions are treated as "first-class" entities, *i.e.,* they may be produced as the result of arbitrary computations and stored inside data structures, just like any other kind of data. As a consequence of being first-class, functions are also "higher-order," *i.e.,* they can take functions as arguments and return functions as results.

Unlike functions, modules are not treated as first-class entities in most dialects of the ML module

[6] See Chapters 2 and 3 of Lillibridge's thesis for more discussion and examples of this [45].

system. They are "second-class" in the sense that the module language exists on a separate plane from the so-called "core" language of ML. Modules may not be passed as arguments to or results from core-language functions, nor can they be stored in data structures. In some sense this is justified by thinking about modules as primarily serving to structure code in a pre-existing core language.

A less defensible aspect of the Standard ML module system in particular—and one not shared by all dialects of ML—is that functors are restricted to be "first-order," meaning that they may only be defined at the top level of the program, not as components of other modules, and thus functors cannot be parameterized over other functors or return other functors as results. It is difficult to explain why functions at the module level of SML are restricted in a way that functions at the core level are not. As a consequence, whether or not they treat modules in general as first- or second-class, most modern variants of ML—including most implementations of Standard ML—do provide support for higher-order functors. The semantics of higher-order functors, however, is an axis in the design space of modules along which we will find considerable variety.

1.2.3 Harper and Lillibridge's First-Class Modules

Harper and Lillibridge's "translucent sums" calculus does not distinguish the language of modules from the core language of terms, thus treating modules as first-class values. The fusion of the module and term levels leads to a pleasantly economical design, in which structures are merely records, and functors are merely functions. In addition, the first-class status of modules allows one to choose between different implementations of an abstract data type at run time based on information that may only be available dynamically.

To steal an example from Lillibridge's thesis [45], suppose one is defining a module implementing dictionaries. Depending on the size of the dictionaries that one will be creating, one may wish to use different implementations. For large dictionaries, a hash table implementation may be appropriate, but for small ones, a linked list implementation will be more space-efficient. If the size is not known statically, first-class modules enable one to make this choice at run time by defining the dictionary module with a conditional expression:

```
structure Dictionary = if n < 20 then LinkedList else HashTable
```

At the same time, however, merging the core and module levels also complicates the type structure of the core language, interfusing it with notions of dependent types and subtyping. As a result, typechecking in the Harper-Lillibridge system is proven undecidable, and moreover it is not clear how ML-style type inference could be adapted to it. For the moment, though, I will ignore the more practical problems with the Harper-Lillibridge approach, in favor of exposing a lack of expressiveness with respect to higher-order functors.

Since functors are just functions in the Harper-Lillibridge system, they are naturally higher-order. Consider, however, a simple canonical example of a higher-order functor, namely the `Apply` functor shown in Figure 1.7. `Apply` takes a functor argument `F` of signature `SIG -> SIG` and a structure argument `X` of signature `SIG`—where `SIG` is a signature with an opaque specification of some type `t`—and it applies `F` to `X`. Ideally, `Apply(F)(X)` should be semantically indistinguishable from `F(X)`. Unfortunately, this turns out not to be the case.

First of all, what is the type (*i.e.,* signature) of `Apply`? The obvious answer—and the one given in Harper and Lillibridge's system—is `(SIG -> SIG) -> (SIG -> SIG)`. Given this type, if we instantiate `Apply` with any arguments of the appropriate types, regardless of what they are, we get out a structure of signature `SIG`. In particular, if we apply `Apply` to the identity functor `Ident`, also defined in Figure 1.7, and a particular structure `Arg` of type `SIG`, then the result `Res1` has type `SIG` as well, giving us no indication that its type component `t` is in fact equal to `Arg.t`.

```
signature SIG = sig type t ... end

functor Apply (F : SIG -> SIG) (X : SIG) = F(X)
functor Ident (X : SIG) = X

structure Res1 = Apply(Ident)(Arg)
(* Res1.t ≠ Arg.t *)
structure Res2 = Ident(Arg)
(* Res2.t = Arg.t *)
```

Figure 1.7: Higher-Order Functor Example

On the other hand, consider what happens when we apply `Ident` to `Arg` directly and bind the result to `Res2`. Although `Ident` does indeed match the type `SIG -> SIG`, its principal type is `(X:SIG) -> (SIG where type t = X.t)`. (This is a dependent function type, where the argument `X` is bound in the right-hand side of the arrow.) Thus, substituting `Arg` for `X`, we see that the type of `Res2` is `SIG where type t = Arg.t`, from which we can infer that `Res2.t` = `Arg.t`. In order to observe this equivalence for `Res1`, we would need to require that `Apply`'s functor argument have the more specific type of `Ident`, but that would in turn place a rather arbitrary restriction on the potential arguments to `Apply`.

1.2.4 SML/NJ's Higher-Order Functors

One approach to remedying this problem was proposed by MacQueen and Tofte [49] and incorporated into the SML/NJ compiler. Their solution is to "re-typecheck" the body of the `Apply` functor at every application site, exploiting knowledge of `Apply`'s actual arguments to propagate more type information. Thus, under their semantics, typechecking the `Res1` module in Figure 1.7 prompts a re-typechecking of `Apply` given the knowledge that `F` in this instance has a more specific type, namely the principal type of `Ident`. Given this added information, the typechecker can then observe that `Res1.t` = `Arg.t`.[7]

MacQueen and Tofte essentially argue that since ML's signature language is too weak to express the dependency between the result of `Apply` and its argument, one must inspect the implementation instead. This is a sensible argument when one has access to `Apply`'s implementation. In the context of separate compilation, however, it is inapplicable, as `Apply`'s implementation may not be available. Moreover, the MacQueen-Tofte solution is fundamentally non-type-theoretic, in the sense that signatures in their language do not encapsulate the information about a higher-order functor that may be needed during typechecking.[8] As I am ultimately concerned with developing an account of ML modules that can be formalized in type theory and that makes sense in the presence of separate compilation, I will focus attention in this thesis on the following alternative approach to higher-order functors.

[7] In practice, the SML/NJ compiler does not actually re-typecheck the body of a higher-order functor every time it is applied. Rather, it employs an implementation technique that mimics re-typechecking without actually doing it. This technique, described by Crégut and MacQueen [7], produces a static representation for each functor that contains all information concerning the "compile-time behavior" of the functor.

[8] In the SML/NJ implementation, the static representation of a functor (described in the previous footnote) *does* encapsulate all information needed about it during typechecking. However, this static representation may not correspond to any signature that the ML programmer can write. Furthermore, Crégut and MacQueen do not provide any formal semantics for it [7].

1.2.5 Leroy's Applicative Functors

Leroy's "manifest types" calculus, while second-class, suffers from the same problem as Harper and Lillibridge's with respect to poor propagation of type information in higher-order functors like `Apply`. In a follow-up paper the following year, however, Leroy [43] presents a solution to the problem that is quite different from SML/NJ's. He proposes an "applicative" semantics for functors as an alternative to Standard ML's "generative" semantics.

Functors in SML, as well as in the translucent sum and manifest type calculi, behave "generatively," in the sense that every time a functor is applied it generates fresh abstract types. In other words, if a functor `F` is applied to the same argument twice, and the results are bound to `A` and `B`, then `A.t` and `B.t` will be considered distinct for any type component `t` that is specified opaquely in the result signature of `F`. For example, since the argument `F` of the `Apply` functor has signature `SIG -> SIG`, the application of `F` in the body of `Apply` results in a module with a fresh abstract type `t`. According to this generative semantics, it makes sense that `Res1.t` is distinct from `Arg.t`, because every application of `Apply` produces a new type `t`, distinct from all others. Nevertheless, as I have argued, this is not the desired behavior for the `Apply` functor.

Leroy proposes instead that, when a functor is applied to the same argument module more than once, it should produce the same abstract types in each result module. In order to realize this "applicative" semantics for functors, Leroy extends the dot notation so that, in addition to projecting types from named structures, one can project types from functor applications.

For example, given that `F` has signature `SIG -> SIG`, the principal signature for `F(X)` in Leroy's applicative functor calculus is `SIG where type t = F(X).t`, which indicates that the type `t` in the result of `Apply` is precisely the one obtained by applying `F` to `X`. Thus, substituting `Ident` for `F` and `Arg` for `X`, we see that `Res1` (defined as `Apply(Ident)(Arg)`) can under Leroy's semantics be given the signature `SIG where type t = Ident(Arg).t`. The signature of `Ident` allows us, in turn, to observe that `Ident(Arg).t` = `Arg.t`, so that `Res1.t` = `Arg.t` as desired.

It is natural to ask whether Leroy's solution carries over to the setting of a first-class module system like Harper and Lillibridge's. The answer is that it does not, and the reason is that in a first-class module system it does not in general make sense to write a type like `F(X).t`. For instance, in the Harper-Lillibridge system, a functor of signature `SIG -> SIG`, when applied, may very well consult some dynamically changing condition—*e.g.,* whether a mouse button is pressed—and the identity of the type components in the module it returns may depend on that condition. Thus, one evaluation of `F(X)` may result in a module whose `t` component is defined to be `int` and another may result in one whose `t` component is defined to be `string`. Since the evaluation of `F(X)` does not always produce the same `t` component, it is senseless to refer to *the* type `F(X).t`.

This is not an issue, however, for a second-class system like Leroy's, which obeys the principle of *phase separation*. A module system obeying phase separation is one in which every module can be split into a static part (comprising its type components) and a dynamic part (comprising its term components), such that the static part does not depend on the dynamic part. In such a system, the type components of modules cannot depend on any dynamic conditions—they are the same every time the module is evaluated. Phase separation is ensured in the case of Leroy's system by the restricted "second-class" nature of the language in which modules are written. A consequence of phase separation is that it makes sense to talk about *the* type `F(X).t` because the type components of `F(X)` are guaranteed to be the same every time it is evaluated.

Although Leroy's calculus, which serves as the basis of the Objective Caml module system, has succeeded in popularizing the idea of applicative functors, both the concepts of phase separation and applicative functor semantics were actually introduced in earlier work by Harper, Mitchell and Moggi [30]. While their calculus admittedly lacks any account of sealing or translucency, it has

```
signature SYMBOL_TABLE =
  sig
    type symbol
    val string2symbol : string -> symbol
    val symbol2string : symbol -> string
    ...
  end

functor SymbolTable () =
  struct
    type symbol = int
    val table : HashTable.t =
      (* allocate internal hash table *)
      HashTable.create (initial size, NONE)
    fun string2symbol x =
      (* lookup (or insert) x *) ...
    fun symbol2string n =
      (case HashTable.lookup (table, n) of
         SOME x => x
       | NONE => raise (Fail "bad symbol"))
    ...
  end
  :> SYMBOL_TABLE

structure ST1 = SymbolTable()
structure ST2 = SymbolTable()
```

Figure 1.8: Symbol Table Functor Example

been a strong influence on other module languages, not least on the design of my own type system for modules. I will discuss their calculus' relationship to mine in more detail in Section 2.2.3.

1.2.6 The Importance of Generativity

The discussion so far might lead one to the conclusion that the applicative semantics for functors is a clear improvement over the generative semantics, but this is not the case—the two are incomparable. As we have seen, the applicative semantics allows for the desired propagation of type information in higher-order functors. For other kinds of functors, however, generativity is essential for guaranteeing the desired degree of data abstraction.

Consider, for example, the `SymbolTable` functor shown in Figure 1.8, which takes no arguments but, when applied, generates a module implementing a symbol table as a hash table. The module represents symbols as integer indices into the hash table, and thus defines the `symbol` type to be `int`. It defines the hash table itself by invoking the `create` function from the standard library `HashTable` module and binding the result to `table`. It then defines two functions for converting between strings and symbols: `string2symbol`, which inserts a string into the table and returns the corresponding symbol, and `symbol2string`, which looks up a symbol in the table and returns the corresponding string. The latter function raises a `Fail` exception if the given symbol is invalid.

Finally, the body of the `SymbolTable` functor is sealed with the `SYMBOL_TABLE` signature. The sealing serves two purposes. One is to prevent the actual `table` from being exported, so that the implementation in terms of a hash table is not revealed. The other is to prevent the clients of the functor from being able to observe that `symbol` is equal to `int` and attempting to pass off arbitrary integers as valid indices into the hash table.

The intention of this implementation is that the `Fail` exception should never be raised because the only values of type `symbol` that clients should ever have access to are those obtained through calls to `string2symbol`, which are clearly valid symbols. Under an applicative functor semantics, however, this intention will not be upheld. Specifically, suppose that structures `ST1` and `ST2` are both defined by calls to `SymbolTable`, as shown in Figure 1.8. According to the applicative semantics, `ST1.symbol` = `ST2.symbol` because both types are equal to `SymbolTable().symbol`. As a result, symbols generated by calls to `ST1.string2symbol` may be passed as arguments to `ST2.symbol2string`, even though such symbols are not necessarily valid indices into `ST2`'s hash table and may cause its `symbol2string` function to raise the `Fail` exception. Therefore, although it is perfectly sound to consider the `SymbolTable` functor applicative, the functor *ought* to be considered generative. Every `symbol` type it produces classifies valid indices into a newly generated symbol table and is thus semantically incompatible with every other `symbol` type.

On the other hand, for a functor like the `Set` functor defined in Figure 1.6, the applicative semantics is perfectly appropriate. Suppose we apply `Set` to the same item module—*e.g.,* `IntItem`—twice and bind the results to `IntSet1` and `IntSet2`. The types `IntSet1.set` and `IntSet2.set` both describe sets of integers ordered according to the same `IntItem.compare` function, so there is no reason to distinguish them.

1.2.7 Supporting Both Applicative and Generative Functors

Each of the major dialects of ML, SML and O'Caml, supports only one semantics for functors: generative *or* applicative, but not both. The analysis above, however, suggests that since each semantics is appropriate in different circumstances, it would be preferable to have a module language that does support both.

The module system of the Moscow ML dialect [56], based to a large extent on Russo's thesis work [65], represents one such hybrid design. In Moscow ML, the programmer can choose, when defining a functor, whether it should behave applicatively or generatively. While the simplicity of this approach is appealing, it is semantically problematic. In particular, one would expect that every application of a generative functor produces distinct abstract types, but this is not the case. For example, if one were to define the `SymbolTable` functor from Figure 1.8 in Moscow ML and label it generative, there would be nothing to prevent one from defining another functor `SymbolTable'` as the eta-expansion of `SymbolTable`—*i.e.,* `functor SymbolTable'() = SymbolTable()`—and then labeling `SymbolTable'` as applicative. Consequently, different instances of `SymbolTable'` would be considered to have equivalent `symbol` types, even though "under the hood" they are really different instances of `SymbolTable`, which according to generativity should have distinct `symbol` types. What is even more troublesome is that the ability to subvert the generativity of `SymbolTable` in this way can be further exploited to break the soundness of the type system [9].

The problems with Moscow ML suggest that in order to guarantee that generativity is respected by the type system, one must restrict the class of functors that may behave applicatively. This is precisely what Shao does in his type system for ML-style modules [69]. Like Moscow ML, Shao's calculus supports both applicative and generative functors. The key idea in Shao's system is to only allow functors to be treated as applicative if their bodies are transparent. (Correspondingly, Shao

refers to applicative functors as "transparent" and generative functors as "opaque.") As a result, the eta-expansion of `SymbolTable` from the previous paragraph could not be labeled as applicative in Shao's system because the principal signature of its body is `SYMBOL_TABLE`, which specifies the `symbol` type opaquely.

Although Shao's approach ensures that generativity is respected, I argue that it is overly restrictive. For example, the `Set` functor from Figure 1.6 could not be treated as applicative in Shao's system, at least not as written, because its body is sealed with an abstract interface. In the case of our implementation of sets as lists, we can work around this problem by hoisting the sealing outside of the functor. In other words, we can leave the body of the functor alone and instead seal the functor itself with the signature

```
(Item : COMPARABLE) -> SET where type item = Item.item
```

Since the sealing no longer occurs *inside* the functor body, the body has a transparent signature, and thus Shao's system will treat the functor as applicative.

However, this technique does not always apply. For instance, suppose that we define a new functor implementation of sets, `Set'`, in which the `set` type is defined by an ML `datatype` declaration instead of as an abbreviation for `item list`. Types defined by `datatype` declarations in ML are abstract and distinct from all other types. Therefore, even without explicitly sealing it, the body of the `Set'` functor will contain an opaque `set` type, and Shao's type system will treat `Set'` as generative. Semantically speaking, however, there is no reason why `Set'` should not behave applicatively, since repeated application of `Set'` to the same item module produces modules with perfectly compatible `set` types.

To summarize, whereas Moscow ML allows one to write too many applicative functors, Shao's language allows one to write too few.

1.2.8 Notions of Module Equivalence

In the above discussion, I have defined applicative semantics of functors informally by saying that a functor behaves applicatively if, when applied to the same module twice, it produces results with equivalent type components. I have implicitly taken for granted in this definition that "the same module" has some agreed-upon meaning. In fact, however, the manner in which equivalence of modules is defined is yet another axis along which several variants of the ML module system differ.

In Leroy's applicative functor calculus (and hence in O'Caml), module equivalence is *syntactic*: two modules are equivalent only if they have the same name. For example, module `X` is equivalent to itself but not to any other module `Y`, even if `Y` is defined by the module binding `structure Y = X`. Consequently, supposing that functor `F` returns an opaque type `t` in its result, then `F(X).t` is equivalent to itself, but not to `F(Y).t`. This is unfortunate, as bindings like `structure Y = X` are commonly used in ML programming in order to give an abbreviated name to a module that will be frequently referred to. Distinguishing between a module name and its abbreviation, based not on any semantic distinction but on a purely syntactic consideration, makes for a somewhat brittle semantics.[9]

Connected to Leroy's syntactic characterization of module equivalence is his requirement that functor applications appearing inside types be in "named form." For example, `F(X).t` is a well-formed type, but `F(struct ... end).t` is not. This named form restriction is useful in order to avoid having the well-formedness of a program depend on the syntactic equivalence of arbitrary module expressions, which is a rather fragile property.

[9] See Section 4.2.6 for another example of peculiar behavior due to the syntactic nature of O'Caml typechecking.

A consequence of the restriction, however, is that there are some higher-order functors which cannot be given fully expressive signatures in O'Caml. For instance, to take an example from Leroy [43], suppose we tweak the `Apply` functor so that instead of returning `F(X)`, it returns `F(struct type t = X.t * X.t; val x = (X.x,X.x) end)`. Due to the named form restriction, the best result signature we can give to this version of the `Apply` functor is the opaque `SIG`, which is precisely how the functor body would be classified in the absence of applicative functors. Thus, in some cases at least, the named form restriction defeats the purpose of introducing applicative functors in the first place.

An approach several people have suggested for remedying this problem is to abandon the named form restriction and employ a more semantic definition of module equivalence. But which definition is best? One is full observational equivalence, or some conservative approximation thereof, but such a definition complicates the type structure significantly by making type equivalence depend on a notion of term equivalence. An alternative, which is employed by Shao [69], Russo [65], and Harper, Mitchell and Moggi [30] in their respective module languages, is to treat module equivalence as purely "static," meaning that it only looks at the type components of modules, not their value components, and thus deems two modules equivalent if their type components are equivalent.[10]

Static module equivalence is sensible in the presence of *phase separation*, discussed above in Section 1.2.5. If modules obey phase separation, then the identity of a type of the form `F(M).t`, even where M is an arbitrary module *expression* (such as `struct ... end`), depends only on the static parts of `F` and M. As the static part of `F` is clearly equivalent to itself, the equivalence of `F(M).t` and `F(N).t` may be decided just by looking at the static parts of M and N, *i.e.,* by comparing M and N according to a notion of static module equivalence.

In addition to avoiding the need for truly dependent types, static module equivalence is the most liberal notion of module equivalence that is still sound. While Shao and Russo both take it as axiomatic that this implies that static module equivalence is the "right" notion, I would argue that this is not necessarily the case. For an example, let us return once again to our trusty `Set` functor. Say that we apply `Set` to two different item modules, which both define the type `item` to be `int`, but which provide different integer comparison functions. According to static module equivalence, the `set` types in the resulting modules should be equivalent, because they result from applying the same `Set` functor to modules whose type components are equivalent. In fact, however, the resulting `set` types are *not* compatible because they describe sets ordered in different ways. Sets constructed from one module will not necessarily meet the representation invariants assumed by the operations of the other module. Thus, treating the two `set` types as equivalent is clearly, in some sense, a violation of data abstraction. It is not a complete violation of data abstraction, though, because the implementation of sets as lists remains hidden to clients of the `Set` functor. So what kind of violation is it? One of the key contributions of the following chapter is to give a clear and satisfying answer to this question.

1.2.9 Conclusion

The goal of this section has been to give the reader a sense of the key questions that arise in the design of the ML module system, as well as some of the answers that have been proposed. If the debates about first-class vs. second-class modules, applicative vs. generative functor semantics, and syntactic vs. static module equivalence leave one's head spinning with tradeoffs, that is a completely natural response to the diverse, fragmented state of the module system literature. The next chapter should hopefully provide an antidote.

[10] The manner in which one extracts the "type components" of arbitrary module expressions (including functors) is called "phase-splitting" and will be made precise in Chapter 4.