

Chapter 2

A Unifying Account of ML Modules

The previous chapter gave a brief survey of several concepts that stand as key points of contention in the design of the ML module system, including applicative functors, generativity, and first-class modules. In this chapter I will go deeper, in an attempt to understand from first principles how the data abstraction afforded by the ML module system actually works. This analysis will produce a unifying account of ML modules within which the differences between existing dialects can be more coherently understood. This account will then guide the design of my type system for ML modules, which I describe at a high level in Section 2.2 and more formally in Chapters 3 and 4.

2.1 An Analysis of ML-Style Modularity

As illustrated in a number of examples from Chapter 1, a central feature of ML modules that distinguishes them from modularity mechanisms in most other languages is that they contain *type* components. Each type component of a module may be exported either with or without its definition (that is, “opaquely” or “transparently”). Furthermore, the type components of a module may be projected out from it in order to form type expressions such as `IntSet.set`.

The analysis in this section is centered around the following simple question:

From which modules should one be allowed to project out the type components?

As we have already seen, this is a question to which different variants of the ML module system give different answers. Standard ML, for instance, restricts the answer to modules that are in named form (*e.g.*, `A.B.C`), whereas O’Caml extends SML’s answer to include functor applications whose constituent expressions are in named form, like `Set(IntItem)`. Shao and Russo extend this further to allow projections from general module expressions such as `Set(struct ... end)`.

The question is one that arises naturally in the design of a module system. It is typically addressed, however, not in its own right, but only insofar as is necessary in order to bolster other design decisions, such as whether the module language is to support applicative or generative semantics for functors or whether it is to treat modules as first- or second-class. In this section I will attempt instead to broach the question directly, independent of any particular design goals, in the hope of achieving a more general, more satisfying answer.

2.1.1 Projectibility and Purity

Let us say that a module expression is considered “projectible” if one is permitted to project out its type components in order to form types. Projectibility is not an absolute condition; in designing our

```

structure X = M
...
structure Y = M
...

```

M is projectible $\iff X.t = M.t = Y.t$

Figure 2.1: Scenario Illustrating Consequences of Projectibility

module language we may define it as we like. The goal here is to understand what considerations should inform our definition.

To begin with, is there any reason not to allow all modules to be treated as projectible? Well, if the type $M.t$ is to have any meaning in a call-by-value language, then it is “the type bound to t in the module value resulting from evaluating M .” Since type equivalence is reflexive, it can only make sense to write $M.t$ if we are sure that every time we evaluate M we will in fact get the same type component t in the result, *i.e.*, if we are sure that $M.t$ is really equal to $M.t$. At least in the context of a first-class module language, not every module expression M has this property.¹

For example, consider the following modified version of the first-class module example from Section 1.2.3:

```

if buttonIsSelected() then LinkedList else HashTable      (MGUI)

```

This expression checks whether a button in a GUI has been selected and, based on that information, returns one module implementing dictionaries or another. Assume both `LinkedList` and `HashTable` export some type—let’s call it t —that will serve as the type of dictionaries, and assume as well that each module implements this type differently. The type $M_{\text{GUI}}.t$ does not make any sense. One evaluation of M_{GUI} may occur at a time when the button in the GUI is selected and may thus produce `LinkedList`, while another evaluation may occur at a time when the button in the GUI is not selected, resulting in `HashTable`. As these module values have different bindings for t , the type $M_{\text{GUI}}.t$ is not well-defined.

So we see that there are certain module expressions like M_{GUI} that it does not make sense to treat as projectible. Why is this interesting? Because, for modules that *can* be considered projectible, more propagation of type information is possible. In particular, consider the programming scenario shown in Figure 2.1, which plays such an important, recurring role in my analysis that I present it here in a somewhat generic form. In this scenario, there are two module variables X and Y that are defined in separate places in a program by the same module expression M , which provides a type component t . The point of this scenario is the following: In a call-by-value language like ML, the module variables X and Y will be bound to the module value resulting from the evaluation of M . If M is a projectible expression, then every time it is evaluated we can be assured that the resulting module value contains the same binding for the t component, and thus the types $X.t$ and $Y.t$ can be considered equivalent because they are both equal (by definition) to $M.t$. Conversely, if we were to substitute the non-projectible module expression M_{GUI} (defined above) in place of M , then it would be unsound to treat $X.t$ and $Y.t$ as equivalent types.

¹It should be understood that M here is not necessarily a module variable/name like `IntSet`, but may stand for any *expression* in the language of modules, examples of which we will see shortly. The distinction between module variables and arbitrary module expressions will be denoted by writing the former in **typewriter** font and representing the latter with metavariables like M written in roman font.

In general, how can we decide whether it is *sound* to consider a module expression M projectible? One approach is to require that M be “pure,” *i.e.*, free of all computational effects, in which case each evaluation of M should in fact compute the same module value. The module expression M_{GUI} is not pure because, each time it is evaluated, it consults the state of the GUI; like dereferencing a pointer to a mutable memory cell, this constitutes an effectful operation.

What kinds of module expressions are pure? All module values are clearly pure, including anonymous structure values such as

```
struct type t = int; val x = 3 end (MVAL)
```

as well as module variables like `LinkedList`, since variables are values in a call-by-value language. Projections from pure module expressions are pure as well, such as `M.Substructure` where M is a pure expression.

It is not really necessary, though, for a module expression to be completely pure in order for it to be soundly considered projectible. For example, the structure expression

```
struct type t = int; val x = ref 3 end (MREF)
```

is clearly not pure in the usual sense. Specifically, the binding for the `x` component has the effect of allocating a new memory cell and returning a pointer to it, and every time `ref 3` is evaluated it will return a different pointer. Nonetheless, it is fine to treat M_{REF} as projectible, as its `t` component will always be defined by the same type `int`.

This example illustrates that, for the purpose of gauging whether it is sound to project the type components from a module expression, all that really matters is purity *with respect to the type components*. Let us thus distinguish two notions of purity, “dynamic purity” and “static purity.” A dynamically pure module is completely free of computational effects. A statically pure module, on the other hand, may have computational effects, but the presence of effects does not influence the identities of the module’s type components. Dynamic purity clearly entails static purity, but static purity is all that is required in order for a module to be soundly considered projectible.²

2.1.2 Phase Separation

The astute reader may have noticed that the module expression M_{GUI} shown above is subtly different from Lillibridge’s example, which I presented in Section 1.2.3. The original version is as follows:

```
if n < 20 then LinkedList else HashTable (MDICT)
```

The difference in the conditional test between M_{GUI} and M_{DICT} is a significant one. Instead of querying the state of the GUI, M_{DICT} checks whether `n` is less than 20, which is a pure operation! Consider plugging M_{DICT} in for M in the scenario of Figure 2.1 (where I assume static scoping, so that both copies of M_{DICT} are referring to the same variable `n`.) Variables are values, so although it is impossible to tell statically whether `X.t` and `Y.t` will equal `LinkedList.t` or `HashTable.t`, it is clear that they will both be defined by the same one. This example illustrates that the static purity of an expression may depend on the dynamic purity of its free variables; the static purity of M_{DICT} depends on the dynamic purity of `n`, *i.e.*, that `n` is instantiated by some *value*.

²A previous version of this work (Dreyer *et al.* [12]) used the terms “statically pure” and “dynamically pure” to mean something completely different, an unfortunate artifact of the overloadability of the terms “static” and “dynamic.” For those familiar with the terminology of our previous work, see Section 2.3 for a detailed comparison.

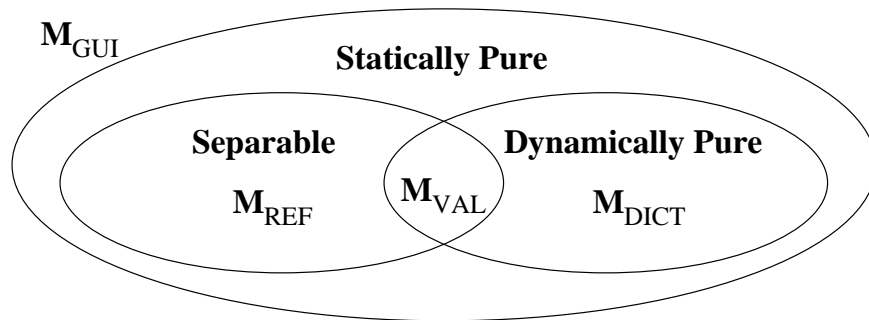


Figure 2.2: Classifications of Module Expressions

My analysis thus far implies that it is sound to treat this pure version of the dictionary module expression as projectible. However, permitting one to project the type \mathbf{t} from this expression means that the language we are designing must support a form of *dependent type*, that is, a type whose identity depends on run-time information (*e.g.*, the value of \mathbf{n}) and cannot be determined statically.³ Dependent types severely complicate the type structure of a language, and ML does not support them. To introduce them vicariously by allowing module expressions like M_{DICT} to be considered projectible would constitute a major extension to the power and complexity of ML, which is not the purpose of the module system.

To avoid the need for true dependent types, one can place an additional requirement on projectible module expressions, which I call “phase-separability” (or “separability,” for short). A module expression is phase-separable if the identities of its type components do not depend, even in a pure manner, on any dynamic values.⁴ Separability ensures not only that a module expression is soundly projectible, but also that its type components are statically well-determined and may thus be projected out without fundamentally expanding the type structure of ML. Of the module expression examples examined above, M_{VAL} and M_{REF} are separable, while M_{GUI} and M_{DICT} are not.

To summarize the discussion so far, Figure 2.2 illustrates the relationships between the classes of statically pure, dynamically pure, and separable module expressions, as well as how the modules M_{GUI} through M_{DICT} fit into the picture. (It is probably worth the reader’s while to stop and check that Figure 2.2 is fully understood before continuing.⁵)

All the variants of the ML module system that I have considered in Chapter 1 require projectible modules to be separable, and the type system for modules that I will describe in Chapters 3 and 4 makes this requirement as well. For the remainder of my high-level analysis, however, I will ignore the practical concern that motivates this requirement, namely the desire to avoid dependent types. I will study how both static purity and separability are preserved (or not preserved) by the features

³One may ask: Aren’t all types of the form $M.t$ dependent types, since M may contain arbitrary code? The answer is no: $M.t$ is not really dependent unless the identity of M ’s \mathbf{t} component relies on the evaluation of code in M , as is the case for $M_{\text{DICT}}.t$.

⁴The term *phase-separable* originates from Harper, Mitchell and Moggi’s view of modules as having a compile-time phase (or static part) and a run-time phase (or dynamic part) [30]. For structures, the static part comprises the type components and the dynamic part comprises the term components. For functors, the static and dynamic parts are a bit trickier to define, but I will do so precisely in Chapter 4.

⁵I leave it as an (easy) exercise to the reader to concoct an example of a module expression that is statically pure, but neither separable nor dynamically pure. The picture in Figure 2.2 makes room for this class of modules, but they do not play an interesting role in the analysis.

of the ML module system, but I will only require projectible modules to be statically pure, not necessarily separable. Tracking both static purity and separability affords us a richer set of module classifications, which in turn allows for a more nuanced account of data abstraction. Precisely what I mean by “more nuanced” will be made clear in the next few sections. By giving ourselves more freedom within this theoretical analysis, we will be able to see more clearly (in Section 2.2.1) what is lost by restricting projectibility in practice to separable modules.

2.1.3 Module Equivalence

This analysis has been driven by the question of how to define projectibility, but an important, closely related question is how to define type equivalence. Suppose we are given two projectible modules M and N , which both provide a type component \mathbf{t} . How do we determine if $M.\mathbf{t} = N.\mathbf{t}$? If the signatures of M and N specify the identity of \mathbf{t} to be transparently equal to types A and B , respectively, then the problem can be reduced to asking whether A and B are equivalent.

Suppose, though, that the signatures of M and N specify \mathbf{t} opaquely. In that case, the answer is that $M.\mathbf{t} = N.\mathbf{t}$ so long as M and N are “statically equivalent,” *i.e.*, they evaluate to modules with equivalent type components. In fact, the notion of static equivalence has already been introduced implicitly in the definition of static purity—statically pure modules are precisely those modules that are statically equivalent to themselves. An alternative, more restrictive notion of module equivalence is “dynamic equivalence.” Let us say that two modules are “dynamically equivalent” if they evaluate to module values that are equivalent both in their type and value components. Dynamic equivalence was implicitly introduced above when I defined the notion of dynamic purity—dynamically pure modules are precisely those modules that are dynamically equivalent to themselves.

Just as the static purity of an expression may depend in general on the dynamic purity of its free variables, static equivalence depends on dynamic equivalence. For example, the module expression M_{DICT} defined above is statically (and dynamically) equivalent to itself, but only under *dynamically* equivalent instantiations of its free variable \mathbf{n} .⁶ The type components of separable module expressions, on the other hand, are by definition indifferent to the dynamic components of their free variables. As a result, a separable module is statically equivalent to itself under instantiations of its free variables that are *statically* equivalent, regardless of whether they are dynamically equivalent.

In order to keep the analysis at a rather informal, intuitive level, I have been deliberately vague about exactly what it means to have “equivalent type components” or “equivalent value components,” assuming some general intuition on the part of the reader. In Chapter 4, we will see a concrete module calculus in which these notions are given formal, albeit syntactic and necessarily conservative, realizations. In the meantime, terms like “separable” and “statically equivalent” should be taken for the conceptual picture they paint, but not for anything more formally semantic.

2.1.4 Total vs. Partial Functors

How do the module properties of purity and separability interact with the mechanisms that are shared by all dialects of the ML module system, namely *sealing* and *functors*? Let us begin with functors.

⁶One can think of the integer \mathbf{n} here as a module that just contains a single value component of type `int`. Correspondingly, any two integers are trivially statically equivalent because they have no type components at all. To be dynamically equivalent, however, they must be equal integers.

To track purity/separability of module expressions in the presence of functors, the chief difficulty is deciding whether a functor application is pure/separable or not, given just the signature of the functor but not its implementation. A common method for tracking purity and effects in the presence of ordinary functions is to distinguish between the types of “total” and “partial” functions, where total functions are those whose bodies are pure and partial functions are those whose bodies are (potentially) impure.

Lifting this idea to the module level, let us distinguish four types of functors, corresponding to the four different module classifications depicted in Figure 2.2:

1. “separably total” functors, whose bodies are separable but may be dynamically impure
2. “dynamically total” functors, whose bodies are dynamically pure but may be inseparable
3. “statically total” functors, whose bodies are statically pure but may be inseparable and/or dynamically impure
4. “partial” functors, whose bodies may be statically impure

It should be clear from this definition that these functor classifications satisfy the same subset relations among themselves as do the properties of separability, dynamic purity, static purity, and static impurity, respectively.

A pleasing property of the total/partial classification is that it corresponds precisely to the distinction between applicative and generative semantics for functors described in Chapter 1. To begin with, say we have a functor variable F whose implementation is compiled separately but whose result signature specifies an opaque type component \mathfrak{t} , and say that we apply F to a separable module expression N .⁷ If F ’s type is known to be separably total, that means the type components of F ’s body are statically well-determined, assuming the type components of its argument are as well. Thus, since N is separable, $F(N)$ is separable, too. On the other hand, if F ’s type is only known to be partial, then $F(N)$ may be impure, let alone inseparable.

Plugging $F(N)$ in for M in our scenario from Figure 2.1, we see that it is sound to consider $X.\mathfrak{t}$ equal to $Y.\mathfrak{t}$ when F is separably total, but potentially unsound to do so when F is partial. In other words, separably total functors behave applicatively, and partial functors behave generatively. It is reassuring that one of the major axes in the design space of modules can be understood simply in terms of total vs. partial functions.

Now what about functors that are statically total, but not separably total? Interestingly, statically total functors also exhibit applicative semantics, but only when applied to *dynamically* pure arguments! To see this, suppose that we have the same scenario as above with F and N , except where F is statically total and N is statically pure. We might imagine that, just as separably total functors take separable arguments to separable results, statically total functors take statically pure arguments to statically pure results. But this is not the case. To see why, let N be the expression

```
if buttonIsSelected()
  then struct val n = 10 end
  else struct val n = 30 end
```

and let F be defined by the declaration

```
functor F(Arg : sig val n : int end) =
  let val n = Arg.n in MDICT end
```

⁷In general, the functor being applied need not be a variable, it may be an arbitrary expression F of functor type. I restrict attention here to the case when F is a variable, mainly for the sake of simplicity, and also because it forces us to look at F ’s interface and not at its implementation to determine whether its application is pure.

<u>If F is:</u>	<u>And N is:</u>	<u>Then F(N) is:</u>
separably total	separable	separable
separably total	statically pure	statically pure
dynamically total	dynamically pure	dynamically pure
statically total	dynamically pure	statically pure

<u>If F is:</u>	<u>And N₁ and N₂ are:</u>	<u>Then F(N₁) and F(N₂) are:</u>
separably total	statically equivalent	statically equivalent
dynamically total	dynamically equivalent	dynamically equivalent
statically total	dynamically equivalent	statically equivalent

Figure 2.3: Semantic Behavior of Different Types of Functors

Recall that `MDICT` tests whether `n` is less than 20 and, depending on the result, returns either `LinkedList` or `HashTable`. This functor `F` is statically total because its body is statically pure. Since the argument `N` does not have any type components, `N` is trivially statically pure as well. However, the application `F(N)` is not pure in any sense; depending on whether the GUI button is selected, it may return `LinkedList` or it may return `HashTable`, which differ in both their static and dynamic components.

The intuitive reason that statically total functors do not preserve the static purity of their arguments is simple. As pointed out in Section 2.1.2, the static purity of an expression may depend on the dynamic purity of its free variables; in particular, the static purity of `MDICT` rests on the dynamic purity of `Arg.n`. Thus, so long as a statically total functor like `F` is applied to a dynamically pure argument (which `N` in this example was not), the result will be statically pure and the functor will behave applicatively. Similarly, it is easy to see that the application of a dynamically total functor to a dynamically pure argument yields a dynamically pure result; but nothing, for instance, can be said about the application of a statically or dynamically total functor to a separable argument.

Figure 2.3 summarizes the behavior of different types of functors on different types of arguments. The table only lists functor-argument pairs for which something positive can be stated about the result. In addition to the cases mentioned so far, the table includes the application of a separably total functor to a statically pure argument. Such an application must produce a statically pure result, because the separability of the functor body ensures that the type components of the result can only depend on the type components of the argument, which by assumption are pure.

Figure 2.3 also describes how total functors preserve *equivalence* of their arguments, which is unsurprisingly in a direct correspondence with how they preserve *purity* of their arguments. For example, just as statically total functors take dynamically pure arguments to statically pure results, they also take dynamically equivalent arguments to statically equivalent results. Clearly, though, a statically total functor like the one defining `F` above will not necessarily produce statically equivalent results given arguments that are merely statically equivalent.

2.1.5 Sealing as a Form of Information Hiding

Now let us turn to the sealing mechanism. Returning to the scenario from Figure 2.1, suppose we plug in for `M` the sealed module expression

```
struct type set = int list ... end :> INT_SET (MSEAL)
```

where the signature `INT_SET` holds the definition of the `set` type abstract. This expression was used to define the `IntSet` module in Chapter 1. As the sealing in M_{SEAL} does not have any actual run-time effect, the evaluation of M_{SEAL} will always result in a module value that defines `set` to be the same type, `int list`. Thus, M_{SEAL} is an example of a separable module expression, and it is perfectly sound to treat it as projectible.

Unfortunately, as the scenario also illustrates, treating M_{SEAL} as projectible has the effect of violating data abstraction! Specifically, `X.set` and `Y.set` will be deemed equivalent, even though nothing about the interface `INT_SET` with which `X`'s and `Y`'s implementations were independently sealed indicates anything that would connect their respective `set` types. In order to make any claim that our type system provides support for data abstraction, we must therefore ensure that sealed module expressions like M_{SEAL} , regardless of whether they are separable, are not considered projectible. Indeed, as one would expect, no actual variants of the ML module system permit such sealed module expressions to appear inside types.

How can we account for this dissonance between separability and projectibility with respect to sealed module expressions? One view is to say that the whole point of sealing is to prevent one from using a module in ways that are perfectly sound, but that violate abstraction. It should therefore not come as a surprise that sealing forces a distinction to be made between the class of projectible modules, from which one is *allowed* to project types, and the class of pure/separable modules, from which it would be *sound* to be able to project types.

Another way to account for the dissonance is to assert that in fact we have made a mistake: sealed module expressions like M_{SEAL} should not even be considered *pure*, let alone projectible. Sealing should render a module expression statically impure, because every time one evaluates a sealed module expression at run time, one generates new and different abstract types. That sealed module expressions must be considered non-projectible then follows as a matter of soundness.

The point of dispute between these views, *i.e.*, the question of whether or not sealing should be treated as an effectful operation, is not merely a pedantic one. It makes a real difference because it affects whether functors that contain sealing in their bodies are deemed total or partial. Under the first interpretation of sealing as a no-op that has no effect on the purity of a module expression, one can employ arbitrary sealing in the body of a functor, and the functor may still be considered total/applicative, so long as its body is otherwise pure. Under an effectful interpretation of sealing, however, any sealing in the body of a functor renders the functor partial/generative because its body is considered impure.

The dispute could be settled quite easily if one of the interpretations led to a semantics for functor-sealing interaction that was clearly preferable, but neither one does. We have already seen examples in Chapter 1 to support this observation. Take the `Set` functor, for example. Its body is sealed, and yet, as I argued in Section 1.2.7, it is appropriate to consider the functor applicative. On the other hand, recall the `SymbolTable` functor from Section 1.2.6. Its body is sealed as well, but it is necessary for the functor to be considered generative in order to guarantee the program invariant that the `Fail` exception will never be raised.

The solution that I propose, then, is to accept that there are multiple varieties of sealing, which are distinguished from one another by how much information they hide about the module being sealed. Figure 2.4 illustrates the semantic effects of several varieties of sealing when applied to a separable module `M`.

The weakest form of sealing, which I call “basic sealing” and denote by `M :> SIG`, seals `M` with the signature `SIG` but does not hide any information about `M`'s separability. The only effect of

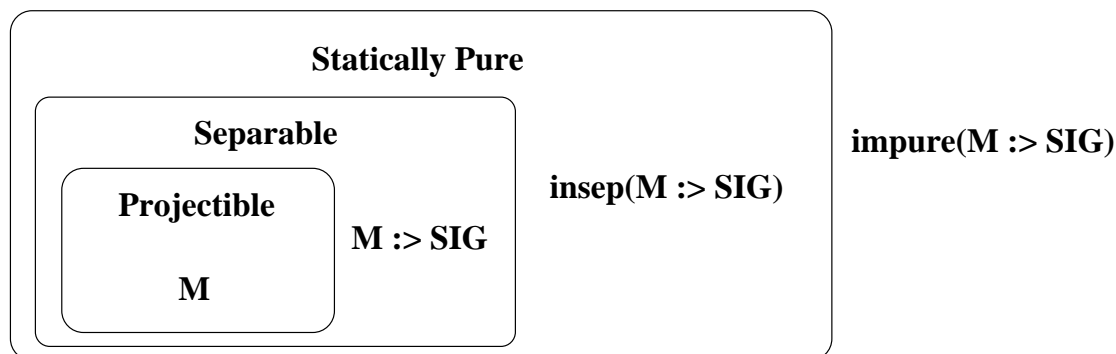


Figure 2.4: Semantic Effects of Sealing

basic sealing is to render the module non-projectible, thus ensuring that the identities of any type components specified opaquely in the signature SIG are held abstract. In contrast, the strongest form of sealing, which I call “impure sealing” and denote by $\text{impure}(M :> \text{SIG})$, not only renders the module non-projectible, but also hides the fact that it is statically pure. Consequently, while basic sealing may be used in the body of a separably total functor, impure sealing may only be used in the body of a partial functor.

While the semantics of the basic and impure forms of sealing correspond precisely to the two interpretations of sealing discussed above, thinking about sealing as a form of information hiding suggests yet another form of sealing in between them, which I call “inseparable sealing” and denote by $\text{insep}(M :> \text{SIG})$. As one might surmise from the name, inseparable sealing hides the fact that M is separable, but does not obscure the knowledge that M is statically or dynamically pure.⁸ As a result, employing inseparable sealing in the body of a functor forces the functor to be considered at best statically or dynamically total, but not separably total.

Each of these three different forms of sealing can be semantically desirable in different circumstances. Consider, for instance, the `SymbolTable` functor. Every time the body of that functor is evaluated, it generates a new hash table in memory and, along with it, a new type of integer symbols that work as valid indices into that particular hash table. Of course, there is no way in ML to define a subtype of integers that only contains valid indices into a hash table. Impure sealing, however, allows us to mimic such a definition. By using impure sealing in the body of the `SymbolTable` functor, we will not only hide the definition of the `symbol` type, but also indicate that the `symbol` type is dependent, at least notionally, on a data structure created at run time by an effectful operation.

What about the `Set` functor? The purpose of sealing its body is not to tie the `set` type to any run-time state, for the `Set` functor is purely functional and has no run-time state. Thus, impure sealing is inappropriate. In writing the `Set` functor, we would really like to define `set` as the type of `Item.item list`’s that are ordered according to the `Item.compare` function. We cannot write such a type definition, of course, because ML lacks dependent types. Inseparable sealing, however, allows us to mimic it. By using inseparable sealing in the body of the `Set` functor, we not only hide the definition of the `set` type, but also “give the impression” that the `set` type depends on

⁸One can imagine another intermediate form of sealing, “dynamically impure sealing,” which hides the fact that the underlying module is dynamically pure—if that is even true in the first place—but does not obscure whether the module is separable or statically pure. I have not included this variety of sealing in the analysis only because I have not yet seen any practical use for it.

the entire functor argument, not only on the `item` type but also on the `compare` function, *i.e.*, that `set` is a *dependent type*. As a consequence, the `Set` functor will be considered statically (but not separably) total, and thus it will only return equivalent `set` types when given dynamically equivalent arguments, *i.e.*, when given equivalent `item` types *and* equivalent `compare` functions.

If we were to use basic sealing instead of inseparable sealing in the body of the `Set` functor, then the functor would be considered separably total and would return statically equivalent results when given merely statically equivalent arguments. As pointed out in Section 1.2.6, this can lead to what amounts to a violation of data abstraction. In particular, if we were to apply the functor to two item modules, `IntLt` and `IntGt`, which both define the `item` type to be `int`, but which provide different comparison functions on integers (one `<` and one `>`), then `Set(IntLt).set` and `Set(IntGt).set` would be deemed equivalent types. They are not, however, semantically compatible types: values of the first type are integer lists ordered by `<` and values of the second type are integer lists ordered by `>`. Although passing values of type `Set(IntLt).set` to the `insert` function from `Set(IntGt)` does not constitute a violation of *soundness*, it does constitute a violation of *abstraction*.

In short, the inseparable and impure forms of sealing are useful for pretending that a type is dependent on a dynamic value or on the result of a dynamic effectful computation, respectively. Conversely, basic sealing is useful when no pretensions of dependent types are required and the sole purpose of the sealing is to hide the identity of a type. One such situation arises when sealing a module that is completely self-contained and does not depend at all on the context in which it is defined. Another example is the type-theoretic interpretation of ML `datatype` declarations given by Harper and Stone [32]. To encode the semantics that each `datatype` declaration in ML generates a distinct abstract type, Harper and Stone translate `datatype` declarations into bindings of sealed modules. These modules have a highly restricted form: they provide a single recursive type, along with two coercion functions, one a “fold” into the recursive type and the other an “unfold” out of the recursive type. While a `datatype` module may depend on other types, it cannot depend on any dynamic values. The purpose of sealing it is solely to hide the implementation of the underlying data type as a recursive type, so the best choice for interpreting `datatype` declarations would be to use basic sealing.

To conclude this discussion, it is worth noting a complaint that is sometimes leveled against both the basic and inseparable forms of sealing, namely that they interact strangely with beta-reduction. For example, if we plug `Set(IntLt)` in for `M` in our scenario from Figure 2.1, then `X.set` will equal `Y.set`. If on the other hand we substitute for `M` the result of *beta-reducing* `Set(IntLt)`, then `X.set` will *not* equal `Y.set`, because each `set` type will result from a separately sealed implementation of sets. As a consequence, in the presence of basic and/or inseparable sealing, beta-reduction of total functor applications is not guaranteed to be type-preserving.

From a methodological standpoint, the interaction of basic and inseparable sealing with beta-reduction is admittedly somewhat unpleasant. I would argue, however, that this deficiency is mitigated by the more accurate propagation of type information that these sealing constructs afford (when the full power of impure sealing is not required). Moreover, as far as type safety of the module language is concerned, the lack of type preservation in the presence of basic and inseparable sealing is not a serious issue—sealing has no actual run-time effect, so we can simply erase all uses of it before executing the program.

2.1.6 Squeezing the Balloon

I have argued that several different levels of information hiding, expressed by different varieties of sealing, are useful and appropriate in different circumstances. In the interest of minimality,

though, it is reasonable to ask whether such an explosion of sealing constructs is truly necessary. In particular, I have motivated the different kinds of sealing in terms of how they force different classifications for the functors in whose bodies they appear—using inseparable sealing in the body of the `Set` functor forces it to be classified as statically total, using impure sealing in the body of the `SymbolTable` functor forces it to be classified as partial, etc. Instead, why not dispense with the multiple forms of sealing, and instead allow the `Set` (resp. `SymbolTable`) functor to be *declared* as statically total (resp. partial) directly?

The answer is that this is a perfectly valid, and essentially equivalent, alternative. If we assume inseparable and impure sealing mechanisms as primitive (as we have), then we can encode a statically total or partial functor declaration as one that implicitly seals its body with the appropriate level of sealing. Conversely, if we assume basic sealing as the only primitive form of sealing, but allow different primitive forms of functor declaration, then we can encode `impure(M :> SIG)` as

```
let partialfunctor F() = M :> SIG in F() end
```

and we can encode `insep(M :> SIG)` analogously using a statically total functor declaration. In short, there is more than one way to squeeze the balloon when it comes to supporting different levels of information hiding. Under the latter approach, however, it is important that *basic* sealing is the one we take as primitive, for that is the weakest form of sealing and it cannot be encoded directly in terms of the other forms of sealing.

The main reason I have chosen to distinguish different forms of sealing is to emphasize the fact that, while each modern variant of the ML module system supports exactly one form of sealing, there is no universally accepted semantics of sealing. In some dialects “sealing” means basic sealing, and in other dialects it means impure sealing. (See Section 2.2.1 for examples.) The semantic framework I have developed here clarifies how the different interpretations of the “sealing” construct relate to one another.

2.1.7 Projectibility and Transparency

Based on the analysis thus far, we can say that a module expression is projectible whenever it is statically pure and free of sealing. This final section of my analysis of ML modularity shows how we may also characterize projectibility in terms of *transparency*. It is based on the following observation: it is fine to treat any module expression as projectible if it has a transparent signature.

The basis for this observation is simple: if a module expression `M` has a transparent signature, then that signature uniquely specifies the identities of `M`’s type components. As the type components of `M` must therefore be the same every time it is evaluated, `M` may at least be considered statically pure. Furthermore, if we plug `M` into the scenario from Figure 2.1, then we see that it doesn’t really matter whether `M` is considered projectible or not because, either way, the transparent definition of `t` in `M`’s signature ensures that `X.t` will equal `Y.t`. Certainly, treating `M` as projectible does not break any abstraction guarantees.

Although it is unclear whether treating transparent modules as projectible serves any practical purpose, the observation that such modules are statically pure is definitely important. For example, suppose there is some functor of partial signature called `F`, and say that we define the following functors that make use of it:

```
functor G (X : SIG) = (struct ... F(X) ... end :> TSIG)
functor OpaqueG (X : SIG) = G(X) :> OSIG
```

Here, `TSIG` is a transparent signature and `OSIG` is a signature with some opaque type specifications. The idea here is that `G`’s body may be impure in the sense that its calls to `F` may have computational

effects and result in the creation of abstract types, but none of these impurities or abstract types are visible to the outside of G because its body is sealed with the transparent signature TSIG . It is therefore safe to treat G as total, and consequently it is safe to treat $\text{Opaque}G$ as total, too.

The reader may wonder: what is the point of $\text{Opaque}G$ in this example? Well, in some sense, whether G is treated as total or partial does not matter very much for G itself—regardless, applications of G will result in modules with equivalent type components because its result signature is transparent. Not so, however, for $\text{Opaque}G$. If G is considered partial, *i.e.*, if we do not observe that transparent modules are pure, then the body of $\text{Opaque}G$ will be considered impure as well, and $\text{Opaque}G$ will be treated as partial. In that case, repeated applications of $\text{Opaque}G$ will *not* result in modules with equivalent type components because $\text{Opaque}G$'s result signature is not transparent. This example illustrates that treating transparent modules as pure is not merely a pedantic issue, it can have an actual effect on the semantics of data abstraction.

We have seen that transparent modules may be considered projectible. What about the converse: can projectible modules always be given transparent signatures? Indeed. To take a simple example, if M is a projectible module with signature `sig type t end`, then clearly M can also be given the transparent signature `sig type t = M.t end`, since M 's t component is certainly equal to $M.t$. In type-theoretic accounts of ML modules, the act of giving a projectible module a transparent signature is often referred to as “signature strengthening” [69] or “selfification” [28]. It is primarily useful in computing a most-precise (or “principal”) signature for M when the identities of M 's type components cannot otherwise be discerned by examining it, *e.g.*, when M is a variable.

In conclusion, I have shown that projectibility and transparency are ultimately equivalent properties. This gives us an alternative perspective on projectibility, but it does not mean that we should abandon our previous characterization of it and use transparency instead as the sole criterion. For certain modules, notably variables, their transparency is predicated on the existing knowledge that they are projectible, not the other way around. Nevertheless, if a module does have a transparent signature, treating it as statically pure is semantically valid and in some cases desirable.

2.2 Fruits of the Analysis

The first section of this chapter has developed informally what one might call an “ML module super-system” in which distinctions are made between several interesting types of module expressions, functors, and sealing mechanisms. This super-system is not itself a language design, but it is useful because it provides a unifying conceptual framework—different dialects of the ML module system may be understood as conservative approximations of the super-system that only recognize certain distinctions and only support certain subsets of features. The conservativity of the existing dialects is due partly to practical concerns such as decidability of typechecking, and partly to actual weaknesses in these dialects that the super-system enables us to articulate more clearly. In Section 2.2.1, I will show how the design points discussed in Chapter 1 may be situated in this chapter's conceptual framework. In Sections 2.2.2 and 2.2.3, I will describe a new module system design that is less conservative and supports more features of the super-system than any of the existing designs while still admitting effective typechecking.

2.2.1 Understanding the Existing ML Module System Designs

Before re-examining the existing ML dialects individually, it is important to note that there is one sense in which they are all conservative—namely, they all require projectible modules to be not just statically pure but also phase-separable. As discussed in Section 2.1.2, this requirement ensures that

types projected from module expressions are normal, non-dependent types. As a result, however, modules like M_{DICT} that are statically pure but inseparable are treated the same as modules like M_{GUI} that are impure. In other words, the distinction between “inseparable” and “impure” becomes moot. Inseparable sealing has the same effect as impure sealing, and the only functor classifications worth tracking are “separably total” and “partial.” (Hence, for the remainder of this section, I will use “total” as shorthand for “separably total.”) Furthermore, static equivalence of separable modules only depends on the *static* equivalence of their free variables, so dynamic equivalence does not play a part in the static semantics of the existing dialects. This makes sense: module equivalence is only relevant to deciding type equivalence; since types are non-dependent in the existing dialects, whether two types are equivalent cannot depend on the equivalence of any dynamic values.

While restricting projectibility to separable modules clearly simplifies the static semantics of the language, it also induces a loss of expressiveness. Specifically, it restricts the sealing mechanism to two variants: basic and inseparable/impure. (The latter variant I will refer to hereafter simply as “impure.”) This is problematic in cases like the `Set` functor, for which, in the full super-system, inseparable sealing provided a superior intermediate choice to the two extremes of basic and impure sealing. In the absence of dependent types, the programmer must choose between one of the less appealing extremes, and neither one is clearly preferable to the other.

First-Class vs. Second-Class Modules Harper and Lillibridge’s first-class module calculus [28] (hereafter, HL) employs a very conservative judgment of separability. Specifically, it considers a module to be separable if and only if it is a value.⁹ The reason for this conservativity is that in HL module expressions are freely intermingled with “core” ML expressions, and it is difficult to track the purity of core ML expressions effectively. Since neither functor applications nor sealed module expressions are values, both kinds of expressions are considered inseparable. In other words, HL treats all functors as partial/generative and only supports the impure form of sealing.

In the remaining variants of the ML module system, the module language is “second-class” in the sense that it exists on a separate layer from the core language and provides a restricted set of constructs. In particular, aside from impure sealing, these second-class languages are so restricted that they do not even provide a way to *write* an inseparable module. (For example, the inseparable module expressions M_{GUI} and M_{DICT} defined earlier in this chapter are not expressible in any of the existing dialects except HL.) It is much easier to accurately decide whether modules are separable in a second-class module system than in a first-class one, because the only source of inseparability is the impure sealing mechanism.

Standard ML Nevertheless, Leroy’s “manifest types” calculus [42], which among the type-theoretic accounts of ML modules is the one that most closely approximates the Standard ML ’97 module system, axiomatizes separability in essentially the same manner as the HL calculus. The only difference is that Leroy, as well as SML, only allows projections of types from modules that are in named form, *i.e.*, variables and projections from variables, also known as “paths.” In terms of actual programming, this does not result in any fundamental loss of expressiveness because types projected from arbitrary module values may always be beta-reduced either to core ML types or to paths. However, given that the SML module language is second-class, its judgment of separability and its assumption that all functors are partial are both unnecessarily conservative.

⁹The notion of value that Harper and Lillibridge use extends the traditional call-by-value notion of value to include projections from values.

Objective Caml Leroy’s “applicative functor” calculus [43] (along with Objective Caml, which is based on it [41]) exhibits one way of addressing this deficiency. It is identical to the manifest types calculus except that, instead of treating all functors as partial/generative, it assumes that all functors are total/applicative. This assumption is only justifiable so long as all modules are separable, which in turn is only true in the absence of impure sealing. Thus, O’Caml’s treatment of all functors as total implies that it only supports the basic form of sealing.

While all modules are separable in O’Caml, not all modules are projectible. O’Caml imposes a named form restriction on projectible modules, which extends SML’s notion of “path” to include functor applications where the functor and argument are both paths. For instance, while both `Set(IntLt)` and `Set(struct ... end)` are considered separable in O’Caml, only the former is considered projectible. One way to account for the named form restriction in the terms of this chapter is to imagine that every structure expression `struct ... end` in O’Caml implicitly contains a sealed submodule defining an abstract type. Hence, the only projectible module expressions are those that do not contain `struct` expressions, *i.e.*, those in named form.

Another notable facet of the O’Caml module system is its notion of *syntactic* module equivalence. The super-system dictates that static equivalence is to be used when comparing the arguments of separably total functors (see Figure 2.3). While syntactic equivalence is indeed a conservative approximation of static equivalence, it is also in many cases a conservative approximation of *dynamic* equivalence. In particular, if we restrict attention to module paths in the SML sense (*i.e.*, not including functor applications), then syntactic equivalence implies dynamic equivalence because SML paths are always values. On the other hand, the O’Caml path `F(X)` is syntactically and statically equivalent to itself, but there is nothing to imply that it is dynamically equivalent to itself because its evaluation may have computational effects. The implication of the sometimes-static/sometimes-dynamic nature of syntactic equivalence is that, when applied to certain arguments, functors in O’Caml behave as if they were separably total, while on other arguments they behave as if they were statically total.

Russo In his thesis, Russo defines two module languages [65]. The first language closely follows SML, supporting only partial/generative functors and impure sealing. The second language, which Russo describes in a chapter on “higher-order modules,” is much like O’Caml, supporting only total/applicative functors and basic sealing. Unlike O’Caml, though, it uses full static equivalence to compare functor arguments. For example, `IntLt` and `IntGt` are considered equivalent in Moscow ML despite having inequivalent value components.

The module system in Moscow ML is a problematic merging of the two module languages from Russo’s thesis—problematic in the sense that it does not correctly track separability. Specifically, as mentioned in Section 1.2.7, the generativity of a partial functor may be defeated in Moscow ML by eta-expanding it into a total/applicative functor. This is clearly a mistaken design, since in general a partial functor cannot soundly be coerced into a total signature. It has also been shown to lead to an unsoundness in the language [9].

Shao Shao’s type system for modules [69] is the only existing design to correctly support both total and partial functors in one language. It only provides one sealing mechanism, however, and it is the impure form of sealing. As a result, when sealing is used in the body of a functor, it forces the functor to be treated as partial/generative. This severely limits the kind of total functors one can write in practice because the presence of any sealed submodule—even a `datatype` declaration—in the body of a functor will be considered an instance of impure sealing and thus render the functor partial. The one exception to this rule is that, when the body of a functor can be given a transparent

signature, Shao’s calculus will allow the functor to be considered total/applicative. As explained in Section 2.1.7, this exception is semantically justified because transparency implies purity. Finally, like Moscow ML, Shao employs full static equivalence when comparing functor arguments.

Summary and Discussion All of the existing dialects of ML described in Chapter 1 take separability as a precondition for projectibility. By doing this, they avoid the need for dependent types, but they conflate the notions of inseparability and impurity as far as typechecking is concerned. Consequently, there are only two kinds of functors recognized by these dialects—separably total and partial—and only two kinds of sealing—basic and impure. The inability to support statically total functors and inseparable sealing is a fundamental limitation of the ML module system which I will not attempt to address in this thesis. Whether these features can be supported without the need for true dependent types remains an open question, one for which I suggest some potential solutions in Chapter 10 on future work.

With the exception of Harper and Lillibridge’s calculus, all the existing dialects treat the module language as second-class, meaning that there is no way to write a module expression that is truly impure or inseparable. There is a basic tradeoff here: a first-class module language like the HL calculus provides more expressive power in terms of the kinds of modules one can write, but it is easier to track separability accurately in second-class languages.

Each existing dialect treats its functors as being either always separably total or always partial, with the exception of Shao’s calculus, which recognizes both kinds of functors. In addition, all the dialects support either the basic or the impure form of sealing, but none supports both. There is no particular reason, however, why a module language could not support both forms of sealing.

Among the dialects in which total functors exist, the arguments to total functors are compared in Russo’s and Shao’s languages using static equivalence and in O’Caml using syntactic equivalence. While the super-system indicates that static equivalence is appropriate when comparing arguments to separably total functors, the O’Caml semantics has the effect of making functors behave as if they were *statically* total on certain common kinds of arguments, namely SML-style paths. For functors like `Set` that we would like to treat as statically total for purposes of data abstraction, the O’Caml semantics at least provides something closer to statically total behavior than the other ML dialects do, but it is still not the real thing. On the downside, as argued in Section 1.2.8, syntactic equivalence is rather brittle in the sense that two modules will be deemed inequivalent even if one is just a renaming of the other (*e.g.*, `structure X = Y`). Furthermore, for functors that the programmer wants to treat as *separably* total, syntactic equivalence is an unnecessarily conservative way of comparing their arguments.

2.2.2 A Unifying Design

In this thesis I propose a new dialect of ML whose module system design is based closely on the super-system set forth in this chapter. This section sketches the high-level design of the language, with comparisons to the existing designs. The following section gives more details regarding the structure of the language definition.

Like the existing ML dialects, my new design avoids the need for dependent types by assuming separability, instead of static purity, as a precondition for projectibility. Consequently, like Shao’s calculus, my language distinguishes between separably total functors and partial functors, but does not recognize the intermediate classification of statically total functors. I compare arguments to total functors using full static equivalence as in Shao’s calculus, because it is less conservative than O’Caml’s syntactic equivalence and more semantically consistent with the super-system. Unlike

any existing dialect, however, my language gives the programmer access to both the basic and impure forms of sealing, each of which we have seen can be useful in different circumstances. It does not support the inseparable form of sealing, but neither does any existing dialect.

With respect to the flexibility of the module language, my new design combines the benefits of first-class and second-class module systems. It is structured like a second-class system in that the language of module expressions is kept separate from that of core ML expressions. It differs from existing second-class dialects, though, in that it allows one to express impure/inseparable modules as well as separable ones. This is accomplished by providing coercions between expressions in the module and core languages. In particular, I introduce a new “package type” $\langle S \rangle$, which is used to classify a module of signature S that has been packaged as (*i.e.*, coerced to the level of) a core-language term. Modules are coerced into the term level by a `pack` operation, and expressions of package type are coerced back to the module level by an `unpack` operation.

For example, the module expressions `MGUI` and `MDICT` defined in Sections 2.1.1 and 2.1.2 would be encoded in my type system as

```
unpack(if ... then (pack LinkedList as SIG) else (pack HashTable as SIG))
```

The module variables `LinkedList` and `HashTable` must first be coerced via `pack` operations to the common package type $\langle \text{SIG} \rangle$, where `SIG` is an opaque signature matched by both implementations; then the result of the core-level conditional expression is coerced back to the module level via `unpacking`. Since the type components of an `unpack`d module expression may depend on the result of a core-level dynamic computation, I conservatively treat all `unpack`d expressions as inseparable.

This hybrid approach offers the practical benefits of second-class systems along with the expressive power of first-class systems. It provides the option of intermingling module and core expressions when so desired. However, compared to a purely first-class language like Harper and Lillibridge’s, it has the advantage that it avoids the insinuation of module-level subtyping into the core ML language, and it enables more accurate tracking of separability for strictly second-class module expressions that do not involve `unpacking`.

2.2.3 A Modular Design

The design I have sketched above will serve as the basis of a new ML dialect, to be defined formally in Part III. As explained in the Introduction, I define this new dialect in the style of Harper and Stone [33]. That is, the programmable “external” language (EL) is defined by an “elaboration” translation into the “internal” language (IL), which is in turn defined by a type system. In the chapters that follow, I will present a slightly simplified version of this IL type system. In order to make the meta-theory a bit less cumbersome, I will omit certain inessential details of the full IL, such as its primitives for exception handling and its treatment of `structure`’s as labeled (instead of unlabeled) records. I will also omit the extensions relevant to recursive modules, which will be studied in Part II, but all the other key features of the language remain. In addition, I will give a decidable typechecking algorithm for this simplified IL that scales straightforwardly to handle the full IL defined in Part III.

As explained above, the IL has two layers: the *core language* of types and terms, and the *module language* of signatures and modules. In existing ML dialects, the module language does not merely serve as a means of “programming in the large,” it also adds fundamental expressive power to the language. For instance, while the core language of Standard ML only supports prenex polymorphism, one can encode a second-class form of higher-kinded and rank-2 polymorphism using

functors.¹⁰ There is good practical justification for sharing the expressive power of the language in this way between the core and module languages, namely that it is necessary to place certain limitations on the power of the core language in order to make ML-style type inference possible. At the level of the explicitly-typed internal language, however, this is not a concern.

Therefore, in the interest of *modularizing* the design of the internal language, I have structured its type system in such a way that all the expressive power of the language is contained within the core language, and the sole purpose of the module language is to provide more convenient mechanisms for structuring core-language code and enforcing data abstraction. Organizing the IL in this way means that the type structure of the core language is self-contained and the module language does not add anything to it. This has several interesting implications.

First, since the language of types is well-defined independently of the module language, there can be no type constructor of the form $M.t!$. Instead, observe that $M.t$ is only a sensible type if M is separable. Separability means precisely that the type components of M may be *separated* out from the rest of the module—there is no way they can really depend on the term components. Correspondingly, in Chapter 4, I define a meta-level function, called for historical reasons $\text{Fst}(M)$, that computes a type constructor representing the “static part” of M . When M is a module variable X , $\text{Fst}(M)$ is a constructor variable X^c , which I take to represent the static part of whatever module will instantiate X . For other module expressions, $\text{Fst}(M)$ is definable in terms of the existing type structure of System F_ω . For example, when M is a structure, $\text{Fst}(M)$ will be a record of type constructors, each of which represents the static part of one of M ’s components. When M is a total functor, $\text{Fst}(M)$ will be a function (at the level of type constructors) that takes as input the static part of M ’s argument and returns as output the static part of its result.¹¹ With this Fst function in hand, we can replace $M.t$ by the core-language type $\text{Fst}(M).t$, which projects the t component from the record of types $\text{Fst}(M)$ representing M ’s static part.

Second, when building the module language, the notion of static module equivalence comes “for free” in the sense that it is definable directly in terms of core-language type equivalence. Two modules M_1 and M_2 are statically equivalent precisely when their static parts $\text{Fst}(M_1)$ and $\text{Fst}(M_2)$ are equivalent type constructors. Since module equivalence is only needed by the type system in order to define type equivalence, it makes sense that the core language should have it built in.

Third, just as the static parts of modules are expressible in the core language of type constructors, the static parts of signatures are correspondingly expressible in the core language of *kinds*. To compute the static part of a signature S , I define another Fst function, this time mapping signatures to kinds, which satisfies the property that whenever M has signature S , $\text{Fst}(M)$ has kind $\text{Fst}(S)$ as well. The definition of Fst on signatures is much as one would expect. In particular, the static part of a structure signature is a record kind describing the static part of each component, and the static part of a total functor signature is an arrow kind. If signatures only contained opaque type specifications, then the kind structure of F_ω would suffice. But how do we faithfully compute the static part of a signature like `sig type t = int end`? That is, how can we ensure that the resulting kind only characterizes the static parts of modules whose t component is `int`?

In order to encode such transparent specifications in the kind language, I employ Stone and Harper’s “singleton” kinds [74]. Whereas kinds in F_ω only provide structural information about the constructors that inhabit them, kinds in the singleton calculus can provide information regarding

¹⁰Specifically, higher-kinded polymorphism can be encoded by parameterizing over a module containing a type component of higher kind, and rank-2 polymorphism can be encoded by parameterizing over a module containing a value component of polymorphic type.

¹¹When M is a *partial* functor, it has no static part because its body is potentially inseparable. Formally, $\text{Fst}(M)$ is defined to be the trivial unit constructor.

the *identity* of their inhabitants. For a type C of base kind \mathbf{T} , the singleton calculus allows us to give it the more precise kind $\mathfrak{S}(C)$, which only classifies types that are equivalent to C . Consequently, when processing a type definition like `type t = C`, we indicate that `t` is shorthand for C by binding it in the context with kind $\mathfrak{S}(C)$.

As singleton kinds make the kind language dependent on the constructor language, the Stone-Harper calculus also supports dependent product and arrow kinds. The kind $\Sigma\alpha:K_1.K_2$ classifies a pair of type constructors with kinds K_1 and K_2 , where K_2 may refer to the first component of the pair via the constructor variable α . Similarly, the kind $\Pi\alpha:K_1.K_2$ classifies a constructor function whose result kind K_2 may depend on the identity of the argument α . For instance, the kind $\Pi\alpha:\mathbf{T}.\mathfrak{S}(\alpha)$ uniquely characterizes the identity function on types.

By combining dependent kinds, singleton kinds, and the standard “opaque” kind \mathbf{T} , the Stone-Harper calculus faithfully models the flexibility of ML’s translucent signatures. For example, the kind $\Sigma\alpha:\mathbf{T}.\mathfrak{S}(\text{int} \times \alpha)$ is a direct analogue of the signature `sig type t; type u = int * t end`. That the two should be in such close correspondence is no coincidence. The Stone-Harper calculus was designed as a target of type-directed compilation for SML, with the goal of preserving the benefits and semantics of translucency in the absence of modules.

One of the major advantages of using the Stone-Harper singleton calculus as the basis of the IL’s core language is that it isolates the meta-theoretic complications of translucency in a language that has been well-studied. Proving that type equivalence is decidable in the presence of singleton kinds is highly non-trivial, but Stone and Harper have already done it. By modularizing the IL so that the module language does not extend the type structure of the core language, I am able to reuse the Stone-Harper meta-theory “off the shelf,” so to speak, greatly easing the burden of proving module typechecking decidable.

It should be noted that organizing the IL in this way is *not* an original idea. It was first propounded by Harper, Mitchell and Moggi in their “phase distinction” calculus [30]. The major difference is that my language supports the modern ML features of translucency and sealing, which theirs, dating back to 1990, does not. Correspondingly, the type structure of their core language does not include singleton kinds. In addition, they treat $\text{Fst}(M)$ as a primitive type constructor, albeit one that is always reducible to some module-free core-language constructor. Treating $\text{Fst}(M)$ as primitive, however, requires them to build a whole equational theory around it. My approach of defining Fst via a meta-level macro is a simpler, more lightweight solution.

Shao [69], in his type system for modules, employs a technique similar to my Fst function for extracting the static parts of modules and signatures (he calls it the “flexroot” function). His calculus is not organized, though, in the fashion that I have advocated: translucency is modeled in his calculus directly at the module level, not through the kind structure, so the equational theory of types is dependent on the module language. As I have argued, I believe my approach makes the language definition more elegant and modular.

The two chapters that follow give the formal definition of the (simplified) IL, Chapter 3 presenting the core language and Chapter 4 the module language.

2.3 Comparison With a Previous Version of This Account

This final section relates the work of this chapter to an earlier, published version by Dreyer, Crary and Harper (hereafter, DCH) [12]. While many of the ideas are the same, the present account makes a more refined set of distinctions than DCH does, and the structure of the IL described in the previous section marks a significant change from (and simplification of) the structure of the DCH module system. There are also some unfortunate clashes of terminology between the two accounts that warrant clarification.

Conceptual Comparison The most important conceptual difference between the two accounts is that the present account makes a distinction between the properties of static purity and separability, while DCH does not. Correspondingly, DCH only observes the distinction between separably total and partial functors, and only recognizes the basic and impure forms of sealing. Although the language design I propose in this thesis suffers from precisely the same limitations as the DCH language, I have attempted to make it clear that this is out of a practical necessity—avoiding dependent types—rather than a semantic one. By starting with a richer set of module classifications and then paring it down, we can better understand what exactly the ML module system is missing—namely, statically total functors and inseparable sealing—and why it is missing them.

In the absence of a distinction between purity and separability, my present account recognizes three module classifications: projectible (and hence separable), separable but not projectible, and inseparable (and hence not projectible). DCH makes precisely these classifications as well, although it describes their genesis in a rather different way. (Actually, DCH makes a fourth classification, but as I explain below it is essentially superfluous.)

The basic tenet of DCH is that a module is projectible if and only if it is *pure*, where the term “pure” means something other than what I have to taken it to mean in this chapter. DCH considers a module expression to be pure if it is free of certain “effects” that result in the creation of new types. There are two kinds of such effects, which DCH terms “dynamic” and “static.”¹²

Dynamic effects are ordinary computational (run-time) effects that affect the identities of a module’s type components, such as the call to `buttonIsSelected` in the module expression `MGUI` from Section 2.1.1. Dynamic effects are also induced, notionally, by sealing a module using impure sealing, which DCH refers to as “strong” sealing. Despite the use of the word “effect,” there are modules that are computationally pure, such as `MDICT` from Section 2.1.2, that DCH considers to be dynamically impure. In short, the meaning of “dynamically impure” in DCH corresponds directly in the present account to “inseparable and possibly impure.” One of the chief advances of the present account over DCH is the observation that there is an important semantic distinction to be made between truly impure modules, like `MGUI`, and pure but inseparable modules, like `MDICT`.

Static effects, on the other hand, are induced by sealing, both by the impure form and by the basic form, which DCH calls “weak” sealing. One can think of static effects as occurring at compile time instead of at run time (hence the name *static*). For this reason, while static effects do render a module non-projectible, they are permitted to occur in the body of a total functor because totality is only a property of the functor’s *dynamic* behavior. In the present account, I have moved away from any discussion of static effects. I now simply point out that data abstraction requires all modules that contain sealing to be treated as non-projectible (unless they are transparent). Since sealing a module using weak/basic sealing does not result in any loss of information about its separability, it is fine for a separable, weakly sealed module to appear inside a total functor.

¹²Do not confuse DCH’s static and dynamic effects with the notions of static and dynamic (im-)purity set forth in this chapter—they mean completely different things! Apologies for any headaches this may cause.

DCH		Classification in the Present Account
Dynamic effects?	Static effects?	
no	no	separable and projectible
no	yes	separable, but not projectible
yes	yes	inseparable, and thus not projectible
yes	no	??

Figure 2.5: Correspondence Between Classifications in DCH and in This Chapter

DCH classifies module expressions based on whether they engender both, one or neither kind of effect. This results in four module classifications, three of which correspond directly to the classifications of the present account. These correspondences are shown in Figure 2.5. The fourth classification, which DCH denotes with the purity classification of \mathcal{S} (for “Statically pure, but dynamically effectful”), is an odd one. Modules that have dynamic effects in DCH are inseparable and must therefore be non-projectible, so what does it matter whether or not they have static effects? Indeed it does not, and I claim this classification is essentially superfluous. It was only included due to a technicality, for more discussion of which see footnote 9 in Section 4.2.2.

Structural Comparison The type system of DCH is structured in the style of traditional type-theoretic accounts of ML modules (excepting Harper, Mitchell and Moggi’s) in the sense that the module language and the type structure of the core language are mutually dependent. In addition, there is an explicit judgment of module equivalence, which is not definable in terms of type equivalence. DCH is similar to my present approach (outlined in Section 2.2.3) in that it employs singletons to model translucency, but instead of singleton kinds it uses singleton *signatures*. The singleton signature $\mathfrak{S}_{\mathcal{S}}(M)$ classifies modules of signature \mathcal{S} that are (statically) equivalent to M . Under my present approach, singleton signatures are definable using singleton kinds, whereas in DCH they are primitive.

There are two main reasons why I prefer the present organization to that of DCH. The first is that lifting singletons to the signature level requires one to reprove much of the Stone-Harper meta-theory in the context of the module language. While not fundamentally difficult, it is nonetheless painstaking work. It is much simpler to use the existing Stone-Harper type system as is.

The second, more subtle reason concerns language extensions. Particularly in the context of recursive modules, one would like to add features to the module language that offer new functionality but do not alter the *type structure* of the language. Under the DCH approach, any change to the module language requires one to go through carefully and check that the new cases do not adversely affect the rather complex proof of decidability of type equivalence. Under my present approach, so long as the “static parts” of any new module and signature constructs are definable in terms of the existing type structure, no theorems regarding decidability of type equivalence will need to be extended or re-examined.

One respect in which the type system I present in the following chapters is not as flexible as DCH’s is that I employ a somewhat more restrictive definition of subtyping for functors. The specific ways in which it is more restrictive are detailed in Section 4.1.2, and the reasons for these restrictions are explained in Section 4.1.4. This does not, however, amount to any fundamental expressiveness gap: for any signatures that DCH considers to be in a subtyping relationship, that relationship may be witnessed in my present type system by an explicit module coercion, which the elaboration algorithm I define in Chapter 9 infers automatically.